**MASTER EN SCIENCES INFORMATIQUES**

# Vers un analyseur statique générique de Java par interprétation abstraite: un analyseur statique simple et un utilitaire d'affichage de l'environnement et du store

Noben, Karl

*Award date:*
2000

FUNDP
INSTITUT D'INFORMATIQUE

# Vers un analyseur statique générique de Java par interprétation abstraite :

## Un analyseur statique simple et un utilitaire d'affichage de l'environnement et du store

- Karl NOBEN -

Mémoire présenté pour l'obtention du
grade de maître en Informatique

RUE GRANDGAGNAGE, 21 • B – 5000 NAMUR (BELGIUM)

Abstract:

*Static analysis of programs covers all programming techniques brought for program optimization and global property checking. Abstract interpretation defines a more formal framework for static analysis, based on so called abstract execution of the program, meaning program is run on a non standard domain of values which covers any possible execution made on standard value sets.*

*The JavAbInt project is in keeping with the general pattern of abstract interpretation, and is applied to Java. The key feature of the project is the design of a generic static analyzer for Java.*

*First steps where achieved with the definition of abstract syntax, concrete semantics and abstract domains by I. Pollet in [Pollet99]*

*This paper refers to a part of the JavAbInt project. Its aim is to generate a very simple static analyzer, based on a lightweight sub language of Java, and to implement a reader friendly graphical interface module that displays the environment and store structure for both the abstract and the concrete domains. This give us an occasion to tackle the boiling domain of graph drawing.*

*Résumé:*

*L'analyse statique couvre l'ensemble des techniques logicielles mises en œuvre pour l'optimisation et la vérification de programmes.*

*L'interprétation abstraite est constituée d'un ensemble de techniques d'analyse statique développées dans un canevas formel. Elle met en œuvre un domaine non standard (dit domaine abstrait) sur lequel seront exécutés les programmes dans le but de couvrir l'ensemble des exécutions possibles sur le domaine standard, et de dériver des propriétés valables pour toutes ces exécutions.*

*Le projet JavAbInt se place dans la droite ligne de l'interprétation abstraite et s'applique au langage Java. Son objectif principal est la mise en œuvre d'un analyseur statique générique pour ce langage.*

*Une première étape a été franchie avec la définition de la syntaxe et de la sémantique opérationnelle d'un sous langage de Java, et la mise en œuvre de domaines abstraits pour ce sous langage par I. Pollet dans [POLLET99]*

*Ce travail reflète un autre aspect du projet JavAbInt. Son objet est double: d'une part la mise en œuvre d'un analyseur syntaxique simple opérant sur un sous langage réduit de Java, d'autre part le développement d'un module d'affichage du couple <environnement, store> tant au niveau de la sémantique abstraite que de la sémantique concrète, en veillant à optimiser la lisibilité de ces structures de données, ce qui nous amènera à aborder le florissant domaine du graph drawing .*

Mes remerciements vont en premier lieu au professeur Baudoin Le Charlier et à Isabelle Pollet pour leur disponibilité, leurs conseils, la passion et la patience avec lesquelles ils transmettent leur savoir.

Mes remerciements vont également aux autres membres du projet – Pascal Van Hentenrijk, Agostino Cortesi, Cécile Hayez et Patrick Hendricks – pour la fructueuse collaboration que nous avons eue.

Un merci tout particulier à mes proches, qui ont accepté que ces trois années ne soient pas des années de trop.

Comment ne pas évoquer Christophe, Samuel, Eric, Christelle, Alice, Vivien, Stéphane, Catherine et D. Métan grâce à qui ces longs mois furent tout sauf austères. A tous, merci.

# Table des matières

# Introduction

## Vers un analyseur statique générique de Java

Le langage Java connaît actuellement de nombreux développements et reçoit auprès de la communauté informatique un écho grandissant. Ses avantages sont nombreux et sont notamment liés à la philosophie « orienté objet » qu'il met en œuvre : des possibilités accrues de réutilisabilité, la facilité de mise en œuvre d'un développement incrémental par l'utilisation de l'héritage permettant à l'utilisateur de faire évoluer son environnement logiciel à son rythme, en toute flexibilité.

Nombre de ces avantages seraient réduits à néant si Java ne reposait pas sur un système de *dynamic binding*. Très puissante, cette technique est source d'une importante inefficacité. Cette perte de performance peut toutefois être atténuée par la mise en œuvre de techniques d'analyse statique ayant pour but l'optimisation du *bytecode*. C'est donc dans cette optique, et plus précisément dans le domaine de l'interprétation abstraite que se place le projet JavAbInt. Ce document vient présenter deux aspects particulier liés au projet.

Dans le chapitre premier, nous préciserons les concepts d'analyse statique et d'interprétation abstraite. Nous verrons également en quoi ces techniques peuvent apporter une amélioration au domaine des langages orientés objet.

Le second chapitre sera l'occasion de présenter une étude de cas. Il y sera question de l'interprétation abstraite d'un sous langage restreint de Java. Après avoir défini ce sous langage, sa syntaxe et sa sémantique, nous présenterons un domaine abstrait trivial sur lequel sera effectué une analyse que nous implémenterons par le biais de l'algorithme multivariant.

Le troisième chapitre traitera de l'affichage d'une représentation graphique des couples <environnement, store> définis par la sémantique du langage, tant au niveau concret qu'au niveau abstrait. Puisque ce problème sera modélisé par l'affichage d'un graphe dans un espace à deux dimensions, nous commencerons par repréciser les concepts fondamentaux de la théorie des graphes.

Dans un second temps, nous présenterons quelques techniques permettant de rendre la représentation graphique d'un graphe la plus lisible possible.

Nous dériverons de ces méthodes un ensemble de critères esthétiques et une méthodologie adaptée à la catégorie de graphe qui nous occupe. Nous terminerons le chapitre par la mise en œuvre de cette méthodologie et une critique sur le résultat obtenu.

Enfin, notons que nous avons volontairement choisi de rédiger ce document dans un style relativement littéraire. Le lecteur se reportera utilement aux références bibliographiques pour de plus amples développements.

# Chapitre 1

## Analyse statique et interprétation abstraite en programmation Orientée Objet : Enjeux et concepts

### 1.1 Enjeux de l'analyse statique

L'analyse statique des programmes étudie les opérations que l'on peut effectuer sur le code source d'un programme afin d'en dégager des propriétés intéressantes, valides pour toutes les exécutions de ce dernier, et ceci sans l'exécuter. A contrario, l'analyse dynamique est mise en œuvre à l'exécution.

Par propriété intéressante, on entend par exemple la conformité à la syntaxe du langage ou, plus utilement, des propriétés sémantiques comme le typage des expressions complexes. L'analyse statique couvre également toutes les techniques ayant trait à l'optimisation des programmes et des codes exécutables.

C'est essentiellement avant l'exécution, au cours de l'implémentation et plus particulièrement lors de la compilation, que la mise en œuvre des techniques d'analyse statique prend tout son sens.

Toutefois, l'analyse statique telle qu'envisagée à ses début fut plus souvent un processus d'optimisation empirique qu'une démarche rigoureuse. L'avènement de méthodes telles l'interprétation abstraite apporte une certaine rigueur, par la mise en place d'une démarche du type mathématique.

## 1.2 Concepts de l'interprétation abstraite

Introduit par P. et R. Cousot [COUS77], le concept d'interprétation abstraite désigne plus particulièrement un canevas mathématique d'analyse statique basée sur l'utilisation – pour chaque programme à analyser – d'abstractions du domaine sur lequel travaille ce programme; ce dernier étant qualifié de domaine concret par opposition au domaine abstrait utilisé dans l'analyse.

Ces abstractions sont intéressantes lorsqu'elles sont définies comme des propriétés du domaine concret, agrégeant au sein de chaque élément du domaine abstrait des ensembles de valeur du domaine concret présentant des caractéristiques communes. Le domaine abstrait aura ainsi fréquemment un nombre fini d'éléments, ce qui permettra de s'assurer que toutes les cas de figures seront évalués. Et d'observer leur comportement lors d'exécutions « abstraites », mettant en jeux ce domaine abstrait.

Même si l'interprétation abstraite induit une perte de précision (potentielle), cet inconvénient est largement compensé par le fait que l'analyse statique est réalisée une fois pour toute alors que ses effets se ressentent lors de chaque utilisation du programme analysé. En outre, un choix d'abstraction judicieux permet bien souvent de réduire cet inconvénient. Le lecteur pourra utilement consulter à ce propos [CORTESI2000]

Reste que pour être utilisable, la mise en œuvre de méthodes d'interprétation abstraites doit être efficace et, en tout état de cause, produire son résultat en un temps fini. Ce rappel est loin d'être superflu dans la mesure ou l'interprétation abstraite flirte en permanence avec l'indécidabilité. Celle-ci étant généralement levée en recourant à des approximations.

L'interprétation abstraite a trouvé dans la programmation déclarative (les paradigmes fonctionnels et logiques) un premier champs d'application. Ainsi l'utilise-t-on pour étudier la propagation des termes clos, l'analyse de *sharing*... Elle a également trouvé d'intéressantes applications dans la programmation orientée objet.

## 1.3 Enjeux de l'interprétation abstraite en programmation Orientée Objet

Nous envisagerons plus particulièrement l'interprétation abstraite dans le cadre du paradigme orienté objet. Celui-ci est entre autre caractérisé par des mécanismes tels que l'héritage, le polymorphisme ou le *dynamic binding* (détermination à l'exécution de la méthode réellement exécutée lors d'un appel). Ces mécanismes rendent la programmation orienté objet très souple, permettant plus de généricité et de modularité. Ainsi, les développeurs qui utilisent ce paradigme ont-ils tendance à multiplier les redéfinitions de classes et à redéfinir les méthodes en fonction des spécificités de chaque classe dérivée. Cela se fait malheureusement au détriment de l'efficacité globale du bytecode généré car la détermination dynamique du code est évidemment plus lourde que la méthode statique.

Dans le domaine de la vérification, l'analyse de types peut permettre d'invalider certains *castings* trop restrictifs, et donc d'éviter des erreurs à l'exécution. Dans ce cas aussi, le type statique trop générique ne permet pas de valider un *casting* alors que l'analyse permet éventuellement de cibler les types dynamiques qui pourraient effectivement entrer en ligne de compte.

Les résultats de ces techniques sont d'une part le remplacement d'appels dynamiques par des appels statiques beaucoup plus efficaces (voir même l'*in lining* du code de certaines méthode à la place d'un appel à celles-ci) et d'autre part, la détection précoce d'erreurs qui n'auraient pas été détectées sans cette technique.

## 1.4 Le projet JavabInt

Le cœur du projet JavabInt (Java Abstract INTerpretation) est l'application des techniques de l'interprétation abstraite à un langage de programmation orienté objet, en l'occurrence Java.

Initié par le professeur B. Le Charlier des Facultés Universitaires Notre Dame de la Paix à Namur et mademoiselle I. Pollet, chercheuse au sein de la même université, le projet voit également son équipe composée du professeur A. Cortesi de l'université Ca'foscari de Venise et du professeur P. Van Hentenrijk de l'Université Catholique de Louvain. En outre, plusieurs étudiants contribuent chaque année au projet par les travaux qu'ils réalisent dans le cadre de leur stage et mémoire de fin d'études.

Les premières réalisations dans le cadre de ce projet, consignées dans le document intitulé « Sémantique opérationnelle et domaines abstraits pour l'analyse statique de Java » sont le fruit du travail d'I. Pollet dans le cadre de sa thèse de D.E.A. Après y avoir défini trois syntaxes abstraites d'un large sous langage de Java, introduisant successivement une information de typage et la labélisation des instructions atomiques, l'auteur envisage deux étapes importantes du processus d'interprétation abstraite : la définition d'une sémantique opérationnelle adaptée à l'analyse des types et la définition de domaines abstraits pertinents pour cette analyse.

La thèse décrite ci dessus servira de cadre théorique aux travaux des mémorands impliqués dans le projet l'année suivante : d'une part la mise en œuvre par C. Hayez et P. Hendricks d'un *parseur* et d'un *type checker* pour le sous langage, d'autre part l'implémentation d'un outil de visualisation de l'état de la mémoire d'un système Java (Environnement et store). Le document que vous avez sous les yeux porte entre autre sur cette dernier partie.

Actuellement, ce projet occupe toujours une place importante dans la recherche menée par B. Le Charlier Et I. Pollet. D'autres étudiants rejoignent l'équipe pour l'année à venir.

Enfin, parallèlement à cela, nous avons réalisé une étude de cas ayant pour objet l'interprétation abstraite d'un sous langage plus réduit encore de Java. Ce dernier conservant toutefois une série de propriétés telles qu'une analyse des types soit toujours possible, avec la possibilité d'en retirer d'intéressants résultats pour le langage dans son ensemble. Il en sera également question dans la suite de ce document.

# Chapitre 2

## Interprétation abstraite : étude de cas.

## 2.1 Méthodologie

L'analyse statique par interprétation abstraite admet un *framework* classique que nous nous sommes efforcés de suivre dans ce chapitre. Outre ce canevas, nous donnons dans ce point les conventions et notations que nous avons adoptées.

### 2.1.1. Définition du langage

Avant de pouvoir réaliser l'interprétation abstraite, il est important de formaliser de manière précise le langage que nous allons étudier. Pour ce faire, nous commencerons par définir sans ambiguïté le (sous) langage sur lequel se fera l'analyse. Cette démarche est intéressante à au moins deux titres : d'une part elle permet de souligner les caractéristiques du langage retenues pour l'analyse, d'autre part elle permet la mise en œuvre de transformations permettant de réécrire dans le sous langage des programmes initialement écrits dans le langage complet.

La forme habituelle de cette définition passe par la formalisation d'une syntaxe concrète, et surtout d'une syntaxe abstraite éventuellement enrichie pour faciliter l'expression de la sémantique.

### 2.1.2. Sémantique du langage

La seconde étape consistera à définir un état général du système, et à clarifier l'influence des diverses instructions sur ce dernier. Pour ce faire, nous définirons la sémantique opérationnelle du langage.

Celle-ci exprime le plus souvent les changements d'état du système sous forme de transitions. Une transition est associée à un « type » d'instruction, et les modifications qu'elle provoque sur l'état du système sont détaillées pour chacune d'entre elles. Chaque exécution étant considérée comme une séquence d'états tels que l'un de ces états est obtenu en tirant une transition déterminée sur l'état précédent, en considérant qu'il existe un état initial et un (des) état(s) terminal(aux).

La sémantique est importante dans la mesure ou elle formalise les fonctionnalités du langage. En outre, nous verrons qu'elle sera très utile lorsqu'il faudra définir une évolution du système abstrait le plus fidèlement possible à l'évolution du système réel.

### 2.1.3. Définition du domaine abstrait « primitif »

Lorsque le langage est défini avec précision, vient le moment de choisir le domaine abstrait. Comme nous l'avons vu au chapitre précédent, ce domaine sera construit sur base de propriétés des valeurs du domaine concret. Il est cependant intéressant de vérifier quelques propriétés de base sur le domaine abstrait. Ces propriétés permettront d'utiliser efficacement les techniques de l'interprétation abstraite :

- Existence d'une relation d'ordre ( notée $\geq$ ) sur le domaine abstrait
- Existence d'une borne supérieure pour tout couple de valeurs du domaine abstrait
- Disponibilité de valeurs abstraites extrémale ($\bot$, l'élément minimal appelé *bottom* et $\top$, l'élément maximal appelé *top*. Ce dernier symbolise la valeur abstraite porteuse de l'information la moins spécifique)
- Fonction d'abstraction permettant de trouver la valeur abstraite associée à une valeur concrète
- Fonction de concrétisation permettant de trouver la (les) valeur(s) concrète(s) associées à une valeur abstraite

Cette dernière fonction n'est pas (forcément) déterministe, en ce sens que la fonction de concrétisation doit théoriquement produire l'ensemble des éléments du domaine concret correspondant à un élément particulier du domaine abstrait. Ceci étant, cela n'est pas toujours possible car les pertes d'informations induites par le mécanismes d'abstraction peuvent survenir. Dans ce cas, nous nous assurons que la fonction de concrétisation donne au moins l'ensemble des valeurs

concrètes associées à un élément abstrait, en acceptant que d'autres valeurs concrètes puissent compléter cet ensemble.

Enfin, notons qu'il existe rarement un seul domaine abstrait permettant de réaliser l'analyse d'une propriété donnée sur les programmes. Toutefois, c'est de la qualité du domaine que dépendra la pertinence des résultats obtenus par l'analyse.

### 2.1.4. Extension du domaine abstrait

Une fois le domaine abstrait défini, il n'est pas immédiatement possible d'effectuer des exécutions abstraites. En effet, l'état du système est composé de diverses structures de données pour lesquelles un pendant abstrait serait le bienvenu, afin de pouvoir réaliser des exécutions abstraites aussi proche que possible des exécutions concrètes. C'est ainsi que classiquement, les structures telles que l'environnement, le store et la pile doivent être traduites et être incorporées au domaine abstrait.

En outre, il est judicieux d'étendre la portée des fonctions d'abstraction et de concrétisation à ces concepts. Ainsi, le passage du concret à l'abstrait, tout comme sa réciproque, sont facilités.

### 2.1.5. Expression d'une sémantique abstraite

L'étape suivante dans la mise en œuvre des techniques d'interprétation abstraite consiste à imiter la sémantique abstraite, mais avec les concepts abstraits. De cette façon, chaque modification de l'état concret par une transition pourra être reflétée par une modification de l'état abstrait au moyen d'une transition abstraite correspondante. Ainsi, l'exécution abstraite mise en œuvre par l'analyse reflétera-t-elle les exécutions concrètes du programme.

La sémantique abstraite se devra donc de définir des transitions abstraites telles qu'une fois tirées sur un état abstrait, elles donne un état ou un ensemble d'états tels que leur concrétisation contienne l'état concret obtenu en tirant la transition concrète correspondante.

### 2.1.6. Exécution abstraite et exploitation des résultats

Enfin, lorsque notre « machine abstraite » est correctement définie, nous pouvons procéder à l'exécution abstraite proprement dite. Il s'agit en fait d'initialiser le système avec l'abstraction de l'état initial du système concret, et d'appliquer ensuite un algorithme dérivant les états successifs sur base des transitions constituant le programme ciblé par l'analyse.

Deux algorithmes peuvent être envisagés pour cela :

- D'une part l'algorithme multivariant. Celui-ci gère deux sous ensembles d'états : l'un contient les états actifs, à savoir ceux pour lesquels on n'a pas encore évalué la possibilité de tirer des transitions ; l'autre contient les états qui ont déjà été inspectés, c'est à dire ceux sur base desquels toutes les transitions possibles ont été tirées.

  S: L'ensemble des états actifs

  R: L'ensemble des états déjà développés

  // Init

  $S := \{< p_0, P_0, (e_0, s_0) >\}$; $R := \{\}$;

  // boucle

  While $S \neq \{\}$ do

  Choisir $< p, P, (e, s) > \in S$;

  $S := S \setminus \{< p, P, (e, s) >\}$;

  $R := R \cup \{< p, P, (e, s) >\}$;

  $S := S \cup (\{< p', P', (e', s') > : < p, P, (e, s) > \longrightarrow < p', P', (e', s') >\} \setminus R)$

  L'initialisation garni l'ensemble S avec l'état correspondant au point de programme initial. L'ensemble R des états inspectés est « évidemment vide au départ.

  L'itération consiste à prendre l'un des états actifs, à la transférer de l'ensemble S à l'ensemble R et à générer tous les états **e'** sur base de l'exécution de toutes les transitions ayant l'état choisi comme état initial.

  Les états **e'** ainsi générés étant versés dans l'ensemble S s'ils ne correspondent pas à de s états déjà traités précédemment, c'est à dire des éléments de l'ensemble R.

  Cet algorithme a l'avantage d'être exhaustif : l'ensemble des états sont générés et testés. Malheureusement, si le nombre d'états générés et mis en attente est élevé (par exemple si le domaine abstrait fait intervenir une approximation trop généreuse), cet algorithme verra son temps d'exécution fortement pénalisé. Le second algorithme que nous présentons réduit cet inconvénient.

- L'algorithme univariant quant à lui travaille sur un nombre réduits d'états et procède par agrégation. Cette réduction du nombre d'états améliore sensiblement les performances de l'analyse, bien que cela induise une approximation et donc une perte d'information supplémentaire.

Vu la taille restreinte des programmes que nous serons amenés à tester, nous avons choisi de mettre en œuvre l'algorithme multivariant. Ce choxi mériterait d'être réexaminé dans la perspective d'une utilisation à plus grande échelle.

Pour une description plus précise de ces algorithmes, le lecteur se reportera aux notes du cours « interprétation abstraite » du professeur B. Le Charlier [INFO3105]

### 2.1.7. Conventions et notations

Nous donnons ici les conventions et notations que nous avons adoptées lorsque nous exprimons formellement les concepts et raisonnements qui apparaissent tout au long de ce chapitre.

- Les noms d'ensembles commencent toujours par une majuscule. Lorsqu'ils sont composés d'une juxtaposition de mots, chacun de ces mots commence par une majuscule

  Ex.: NomMethode

- Les mots clés du langage et les symboles (accolades, parenthèses, point virgule...) sont notés en italique et en gras.

  Ex. : Expr ::= Des | ***null***

  Cond ::= C | Des.***instanceOf(***NomClasse***)***

- La barre verticale ( | ) sépare deux alternatives.

- Les parenthèses définissent la portée de l'un des symboles décrits ci-dessous. Ceux-ci se notent en exposant à la suite de la parenthèse fermante. Lorsqu'aucun symbole n'est présent, les parenthèses signalent simplement l'unité logique des éléments qu'elles encadrent, et attirent l'attention du lecteur sur ce fait.

  Ex. : Instr ::= Des = ***new*** NomClasse***(***(Expr (,Expr)$^{*}$)$^{?}$ ***)*** ***;***

  | Des = Expr ***;***

- Une astérisque ($^{*}$) caractérise une séquence d'éléments, éventuellement vide. Un point d'interrogation ($^{?}$) caractérise un élément facultatif

- En outre, les notations fonctionnelles usuelles sont utilisées.

## 2.2 Syntaxe

### 2.2.1. Syntaxe concrète – définition du sous langage

Comme nous l'avons vu, le projet JavabInt a pour objet l'interprétation abstraite du langage Java, et c'est dans ce cadre qu'il faut replacer les travaux qui sont présentés ici.

Nous avons choisi d'étudier un sous langage très simplifié de Java. Ces simplifications permettent de ne considérer que les aspects directement liés à l'analyse des types, en se garant bien de dénaturer l'essence de Java, notamment en ce qui concerne la façon avec laquelle il met en œuvre les mécanismes de la programmation orientée objet.

Les restrictions suivantes différencient notre sous langage de Java :

- Les types simples ont été supprimés. Bien que constituant une facilité de programmation certaine, ils ne constituent pas un enjeu fondamental dans l'optique de l'analyse statique par interprétation abstraite. L'ensemble des variables correspond donc à des objets dans notre sous langage. On peut simuler les types de bases en définissant des classes dépourvues de champs.

- Les classes ne contiennent pas de membre statique. Cette restriction peut être aisément contournée en rassemblant l'ensemble des méthodes statiques au niveau d'un ou plusieurs objets instanciés au début du programme.

- Un nom de méthode correspond à une et une seule signature. Il n'y a donc pas de conflit pour les noms de méthode, à l'exception bien évidemment de la redéfinition de méthodes héritées. Cette restriction peut être contournée en étendant les noms de méthode Java avec le type des paramètres de la méthode, c'est pourquoi cette restriction ne nous parait pas poser de problèmes.

- Les champs sont tous privés. Cette restriction nous permet de simplifier fortement l'initialisation des environnements lors de l'exécution d'appels de procédure. Elle n'est pas trop restrictive puisque l'utilisation d'accesseurs (méthodes *set* et *get*) permet de la contourner.

- Les méthodes sont toutes publiques.

- Les modificateur d'accès de méthode ou de champ (*private*, *protected*, *public*...) ne sont pas implémentés, vu les deux points précédents.

- Chaque méthode se termine pas l'instruction *return*. Il s'agit simplement d'une simplification permettant d'unifier les méthodes qui retournent un paramètre de celles qui ne le font pas. Nous renforçons cette contrainte en imposant, de plus, que toutes les méthodes renvoient une valeur. Pour ces mêmes raisons, les constructeur se termine par l'instruction "*return this*".

- Chaque classe contient un constructeur unique. Dans la lignée de l'impossibilité d'avoir plusieurs méthodes synonymes au sein de la même classe, cette contrainte peut être contournée en créant un constructeur agrégeant les fonctionnalités des divers constructeurs envisagés, on en se utilisant un constructeur simple complété de méthodes d'initialisation distinctes.

- Le programme contient une et une seule méthode *main*. En outre, nous avons décidé, pour une raison de facilité, d'isoler cette méthode dans une classe qui est spécifique nommée *ClasseMain*.

- En outre, nous avons réduit le panel d'instructions à sept instructions fondamentales. La plupart des autres instructions pouvant être simulées en combinant ces dernières.

  *L'instruction de déclaration de variable*. Elle est réduite à sa forme la plus simple, en ce sens qu'elle ne permet pas l'initialisation avec une valeur particulière. Une affectation peut compenser cette restriction.

  *L'affectation simple.* Comme le langage ne prévoit pas de type de base, nous ne pouvons affecter que des objets. L'appel de méthode est une autre possibilité d'affectation décrite ci-dessous.

  *L'instruction conditionnelle simple* "*if – then – else* " n'admettant comme condition qu'une valeur booléenne ou un test sur le type ( *instance Of* )

  *L'appel de méthode* qui implique obligatoirement l'affectation du résultat à une expression de gauche (variable ou champs de l'instance courante)

  L'appel de constructeur ("new")

  L'instruction de fin de méthode ("return")

  L'appel du constructeur de la classe mère ("super")

La syntaxe concrète est donc décrite comme suit :

Id ::= [a...z,A...Z,0...9]*

NomMeth ::= Id + {***main***}

NomVar, NomChamp, NomClasse ::= Id

DeclChamp ::= NomClasse NomChamp ***;***

DeclVar ::= NomClasse NomVar ***;***

ListParamFormel ::= NomClasse NomVar (, NomClasse NomVar)*

Des ::= NomVar | ***this*** | NomChamp

Expr ::= Des | ***null***

C ::= { True | False | DontKnow }

Cond ::= C | Des.***instanceOf(***NomClasse***)***

Instr ::= Des = ***new*** NomClasse***(***(Expr (,Expr)*)? ***)*** ***;***
| Des = Expr ***;***
| Des = Des.NomMeth ***(*** (Expr (,Expr)*)? ***)*** ***;***
| ***if*** ***(*** Cond ***)*** ***{*** Instr* ***}*** (***else*** ***{*** Instr* ***}***)? ***;***

DeclConstr ::= NomClasse ***(*** ListParamFormel?***)***
***{*** DeclVar*( ***super(*** (Expr (,Expr)*)? ***);*** )? Instr* ***return this ;}***

DeclMeth ::= (NomClasse) NomMeth ***(*** ListParamFormel? ***)*** ***{*** DeclVar* Instr* ***return (*** Expr ***);}***

DefClasse ::= ***class*** NomClasse (***extend*** NomClasse)? ***{*** DeclChamp* DeclConstr DeclMeth* ***}***

MethodeMain ::= ***void main (*** ListParamFormel? ***)*** ***{*** DeclVar* ***Init*** Instr* ***Fin}***

MainClasse ::= ***class ClasseMain {*** MethodeMain ***}***

Prog ::= DefClasse* MainClasse

### 2.2.2. Syntaxe abstraite

La syntaxe concrète est, nous l'avons vu, fort dépouillée. La syntaxe abstraite n'en sera donc que légèrement plus structurée. Nous introduisons toutefois un mécanisme de labélisation. Ce mécanisme sera surtout utiles au niveau de la sémantique, afin d'identifier avec précision les étapes successives de l'exécution du programme.

Intuitivement, le processus de labélisation est un jalonnement de programme, et repose sur la mise en place de jalons qui baliserons l'exécution. L'acception habituelle du terme jalon est associée à des vocables tels que points de contrôle ou de vérification. C'est précisément ce type de mécanisme que nous souhaitons mettre en place. Le jalonnement de la syntaxe permettra donc de définir des points de programmes et, pour chacun de ce point, de contrôler l'état du système.

Si le principe du jalonnement est acquis, encore faut-il savoir quelle en sera la granularité. Puisqu'à priori nous souhaitons effectuer une analyse très fine, il serait de bon ton de placer un nouveau jalon (un point de programme) à la suite de chaque modification élémentaire de l'état du système. Sans entrer dans les détails de ce qu'est cet état du système – qui sera par ailleurs décrit dans la sémantique, lors d'un point suivant – nous pouvons en première approximation reconnaître en chaque instruction une modification de l'état. Contrairement à ce qui est défini dans la syntaxe concrète, on peut également considérer les déclarations de variables comme des modifications de l'état (une nouvelle information enrichit le système), ce qui explique la présence de points de programme dans la définition de ces concepts. Il faut enfin noter que chaque instruction est flanquée d'au moins deux points de programmes : l'un correspondant à l'état du système avant la modification induite par l'instruction, l'(les) autre(s) donnant l'(les) états au terme de l'exécution de celle-ci. L'enchaînement des instructions impliquant qu'un point de programme au terme d'une instruction donnée soit aussi le point de programme à l'origine de l'instruction suivante.

Nous restons volontairement flou en ce qui concerne la forme concrète que prendront les points de programmes. Ils devront simplement vérifier une propriété d'unicité des valeurs afin de caractériser sans ambiguïté un point précis de l'exécution d'un programme. Le lecteur peut par exemple considérer qu'il s'agit d'entiers distincts.

La syntaxe abstraite est donc décrite comme suit :

NomClasse, NomVar, NomChamp ::= Id

NomMeth ::= Id + {***main***}

DeclChamp ::= NomClasse NomChamp

DeclVar : := pt ***var*** NomClasse NomVar pt

Des : := NomVar | ***this*** | NomChamp

Expr : := Des | ***null***

Cond : := c | ***instanceOf*** Des NomClasse

Instr : := pt ***affect*** Des Expr pt
| pt ***new*** Des NomClasse Expr* pt
| pt ***proc*** Des Des NomMeth Expr* pt
| pt ***if*** Cond pt pt

DeclConstr : := NomClasse (NomClasse NomVar)* DeclVar*
(pt ***super*** Expr* pt) ? Instr* (pt ***return this***)

DeclMeth : := NomClasse NomMeth (NomClasse NomVar)* DeclVar* Instr* (pt ***return*** Expr)

DefClasse : := NomClasse (***extend*** NomClasse) ? DeclChamp* DeclConstr DeclMeth*

MethodeMain : := ***main*** (NomClasse NomVar)* DeclVar* ***Init*** Instr* ***Fin***

MainClasse : := MethodeMain

Prog : := DefClasse* MainClasse

## 2.3 Sémantique du langage

### 2.3.1. Sémantique opérationnelle

Nous donnons ici les concepts qui constitueront l'état du système pendant l'exécution. Nous détaillerons également deux transitions particulières. Pour une description complète de la sémantique du sous langage de Java, le lecteur se rapportera utilement à [HHN2000]

L'état du système est Composé de 4 informations :

- Une pile permettant le stockage d'informations au fil des appels de méthodes
- L'environnement courant, associant à chaque nom de variables (et à la référence de l'instance) la location de l'instance contenant l'information accessible depuis cette variable. Intuitivement, la location d'une instance est une référence à cette instance particulière dans l'espace de toutes les instances, un peu comme une adresse permet de retrouver une maison dans l'ensemble de constructions d'une ville ou d'une rue. L'environnement de la ville serait un annuaire associant en temps réel une adresse à chaque nom de propriétaire.
- La mémoire informationnelle (store) telle qu'utilisée à l'instant visé, c'est à dire une association des locations à leur contenu. Dans la métaphore du point précédent, il s'agit d'associer une construction physique à une adresse.
- Une indication du point précis ou l'on se trouve dans l'exécution du programme.

De manière plus formelle, l'état peut être vu comme un quadruplet

IEtat = IPtsProg X IPile X IEnv X Store

Où IEnv = INomVar + {***this***} + {***super***} → ILoc + {***null***} + {***undef***}

Store = ILoc → (INomClasse X IInstance) + {***undef***}

Avec IInstance = INomChamp → ILoc + {***null***} + {***undef***}

IPile = {(υ1, ..., υn) | n ∈ IN ∧ ∀i : 1 ≤ i ≤ n :

υi ∈ (IEnv X IPtsProg X TypeAppel)}

La définition d'environnement est formalisée par une fonction. Un environnement particulier n'associe pas forcément une location à tous les noms de variables (c'est par exemple le cas pour les variables qui ne sont plus accessibles, ou qui n'ont pas encore été déclarées). Ceci explique que l'ensemble d'arrivée soit étendu avec ***undef*** qui sera retourné par l'environnement pour tous les noms de variables non déclarés. ***Null*** sera quant à lui retourné pour les variables déclarées mais qui ne sont pas (encore) associées à une instance.

TypeAppel est l'entité chargée de caractériser de manière univoque la nature de l'expression qui recevra le résultat de l'exécution de la méthode. En l'occurrence, il s'agit du type d'expression (Variable ou champ de l'instance courante) et de l'identifiant de celle-ci (son nom). Ces informations sont nécessaires dans la mesure ou des manipulations différentes sont envisagées au terme de l'exécution d'une méthode selon que le résultat sera affecté à un champ ou une variable.

L'état du système évolue de point de programme en point de programme, au fil des instructions. Cette succession peut n'est pas entièrement déterministe. Elle dépend alors de l'état du système lui même.

L'exécution d'une instruction, ou pour utiliser la terminologie des graphes d'états, le fait de tirer une transition, peut modifier les quatre composantes de l'état : la pile est incrémentée ou décrémentée lors d'appels de méthodes ou au terme de l'exécution d'une d'entre elles ; ces mêmes instructions, ainsi que les déclarations de variables, modifient l'environnement (ajout et/ou suppression de noms de variables); les affectations changent la mémoire du système alors que la quasi totalité des instructions voient leur exécution clôturée par un changement du point de programme courant.

Nous donnons, à titre d'exemple, les modifications de l'état qu'entraînent deux transitions...

*Affectation*

La transition est exprimée comme suit :

{ p } ***affect*** $v_1$ $v_2$ { q }

Et doit être lue comme suit : au point de programme **p**, la transition est l'affectation de la valeur de la variable $\mathbf{v_2}$ à la variable $\mathbf{v_1}$. L'exécution se poursuit ensuite par la transition ayant **q** comme point de programme initial.

Il en découle immédiatement que le quadruplet constituant l'état du système est modifié comme suit :

- Le point de programme courant passe de **p** à **q**
- La pile est inchangée
- Le *store* l'est également. Aucune nouvelle information n'est constituée, seule une référence est modifiée, ce qui se traduit au niveau de l'environnement
- L'environnement est modifié en ce sens que le nom de variable $\mathbf{v_1}$ n'est plus associé à son contenu précédent, mais bien au même contenu que celui référencé par $\mathbf{v_2}$. Celui-ci est obtenu en prenant le contenu de $\mathbf{v_2}$ dans l'environnement initial.

Formellement, cette transition s'exprime donc comme suit :

$$< p, P, (e, s) > \longrightarrow < q, P, (e[v_1/Loc(e, s, v_2)], s) >$$

Les possibilités de modification de l'état du système que nous pouvons envisager sont, comme nous l'avons vu, au nombre de sept. Outre l'affectation, nous présentons encore l'appel de méthode qui représente un aspect intéressant de la sémantique.

*Appel de méthode*

Lors d'un appel de méthode, la nature des modifications de l'état sont de deux types : d'une part il faut ajouter à la pile les informations permettant de restaurer l'état du système lorsque l'exécution de la méthode sera terminée, d'autre part, il faut créer un nouvel environnement, car la portée des variables locales de la méthode appelante trouve ici sa limite, alors que les paramètres passés lors de l'appel doivent être disponibles dans la méthode appelée, et donc apparaître dans le nouvel environnement.

La transition est exprimée comme suit :

{ p } ***proc*** expr_ret v m $v_1$, $v_2$, ..., $v_n$ { q }

Ce qui signifie qu'au point de programme **p**, la transition est l'appel de la méthode **m** sur la variable **v** avec le paramétrage $\mathbf{v_1}$, $\mathbf{v_2}$, ..., $\mathbf{v_n}$. Le résultat de l'exécution de cette méthode sera affecté à la variable ou au champ de l'instance courante **expr_ret**. Au terme de l'exécution de la procédure, le programme se poursuivra par la transition ayant **q** comme point de programme initial. Il faut noter que ce point de programme n'est pas celui qui se trouve au début de la méthode appelée (**r**). Ce dernier est obtenu indirectement, comme nous allons le voir ci dessous.

Le quadruplet constituant l'état du système est modifié comme suit :

- Le point de programme courant passe de p à r. Cette information est obtenue en prenant le premier point de programme de la méthode appelée, dont nous avons le nom **m**. Comme nous savons que les noms de méthodes sont uniques, il « suffit » de récupérer le type dynamique de **v** et d'extraire le premier point de programme de la méthode **m** de la classe du type dynamique de **v**.

- La procédure décrite ci dessus n'est pas tout à fait complète dans la mesure ou dans un langage orienté objet, un objet hérite des méthodes des classes dont il est dérivé, et la méthode peut donc se trouver dans une classe différente de la classe associée au type dynamique de la variable.

- La pile est augmentée d'une couche composée de l'environnement initial **e**, du point de programme **q**, et d'informations sur l'expression qui sera affectée du résultat de l'exécution de la méthode. Ces informations son nécessaire à la restauration de l'état de la méthode appelante au terme de l'exécution de la méthode appelée ainsi qu'a la bonne utilisation du résultat produit par cette dernière.

- Le *store*, par contre n'est pas affecté par un appel de méthode. Certes, certaines location seront inaccessibles puisqu'aucune références (directes ou indirectes) n'y seront faites depuis l'environnement de la méthode appelée, mais au terme de l'exécution de celle-ci, l'information sera à nouveau disponible puisque l'environnement de la procédure appelante sera restauré.

- L'environnement est modifié en profondeur. Au terme de la transition, le système sera « dans » la procédure appelée, ce qui signifie que l'instance courante et (accessible via le mot clé ***this***) sera l'objet référencé par **v** dans l'environnement initial, et que les seules variables accessibles depuis l'environnement seront les paramètres, puisqu'il n'existe pas de variables globales dans notre sous langage. Ces paramètres auront les noms qui sont définis dans la déclaration de la méthode (paramètres formels), au sein de la classe associée au type dynamique de l'instance sur laquelle est exécutée la méthode, mais seront associées aux locations associées par les paramètres effectifs.

Formellement, cette transition s'exprime donc comme suit :

$< p, P, (e, s) > \longrightarrow < r, <e, q, ( \{\textbf{\textit{var}} \mid \textbf{\textit{champ}}\}, expr\_ret)> : :P, (e', s) >$

où expr _ret, $v \in dom(e)$

$r = getPremierPointMethode (m, TypeDynamique(e, s, v))$

$e' = \perp[\textbf{\textit{this}}/Loc(e, s, v), u_1/Loc(e, s, v_1), \ldots u_n/Loc(e, s, v_n)]$

où les $\mathbf{u_i}$ sont les paramètres formels définis dans la déclaration de la méthode **m** appartenant à la classe associée au type dynamique de **v**.

Ce qui signifie que l'état $< p, P, (e, s) >$ est modifié quasi intégralement, seul le store reste inchangé. L'environnement e' est construit à partir d'un environnement vide ($\perp$) ou l'on ajoute un ensemble v1, ..., vn d'éléments ayant pour nom un paramètre formel et pour location celle du paramètre effectif correspondant. Le store est également complété par une référence à la nouvelle instance courante, associée à la location de la variable sur laquelle la méthode est appelée. Le pile est incrémentée d'une nouvelle couche <e, q, ( {***var*** | ***champ***}, expr_ret)>.

### 2.3.2. Sémantique abstraite

Comme nous l'avons vu dans la définition de l'interprétation abstraite, cette technique consiste à réaliser une exécution du programme non sur des valeurs concrètes mais sur des éléments d'un domaine abstrait; éléments représentant les valeurs prises par certaines propriétés sur le domaine concret.

Plus particulièrement, c'est sur les types dynamiques que se porte notre attention. En effet, les mécanismes de *dynamic binding* inhérents à Java introduisent beaucoup de souplesse et de généricité, mais ils ne permettent pas de connaître avec exactitude le type d'une variable. la seule information disponible au moment de la compilation est le type statique.

Prenons le cas d'une méthode déclarée comme prenant un paramètre dont le type statique A est très « générique » (i.e. de nombreuses classes sont dérivées de la classe définie par celui-ci). La manière d'implémenter les fonctionnalités de cette méthode étant différentes en fonction du type du paramètre.

Lors de la compilation de cette méthode, le code machine généré tiendra évidemment compte de cette généricité en prévoyant tous les cas de figure, le sélection s'effectuant à l'aide de force conditions et tests.

Si dans un programme précis, cette méthode est appelée avec un paramètre dont le type dynamique est toujours de la même classe B dérivée de A, on pourra considérer que la généricité de la méthode pénalise l'efficacité générale du programme.

Une des solution consiste à recenser les différentes utilisations qui seront faites de ce paramètre (i.e. les différents types dynamiques qui lui seront attribués au cours de n'importe quelle exécution) et de calculer la borne supérieure de celles-ci. Ce n'est bien sûr pas toujours possibles (parce que cette information n'est pas toujours disponible, et dans ce cas il faut utiliser l'information par défaut), mais cela peut permettre le cas échéant d'optimiser le code machine produit en y retirant les portions de code qui ne seront manifestement jamais atteinte.

L'interprétation abstraite travaille, nous l'avons vu, sur des propriétés du domaine concret. Ce dernier ne se limite pas au domaine des valeurs, mais à toutes les composantes de l'état du système.

#### 2.3.2.1. Etat abstrait

*Store abstrait*

On se souviendra que le store (concret) était une fonction des locations dans les instances. Dans le cas abstrait, la propriété qui nous intéresse est le type des instances plus que leur contenu réel. Il n'est donc pas pertinent de différencier deux instances d'une même classe, et il n'y a plus d'intérêt à avoir l'ensemble des locations comme domaine du store.

Nous réalisons une abstraction sur les instances en mémoire, et donc sur le store, de la façon suivante : l'abstraction de toutes instances concrètes d'une classe $\mathbb{C}$ en une seule instance abstraite.

L'agrégation décrite dans le paragraphe précédent implique que le store abstrait soit une fonction des locations vers les instance abstraite. Sachant qu'il n'y aura qu'une et une seule location par type, on peut aussi exprimer le store abstrait comme une fonction associant types et instances abstraites. Sous ces conditions, la notion de location ne sera guère utilisée explicitement.

Nous avons par ailleurs choisi de ne pas abstraire les points de programme (même si cela aurait pu être fait, par exemple en définissant un point de programme abstrait comme l'agrégation des points de programme concrets précédant toutes les transitions d'un type particulier).

*Environnement abstrait*

L'environnement abstrait associe à un nom de variable un type abstrait (l'abstraction du type de l'instance associée à cette variable par composition de l'environnement et du store).

Il en résulte que contrairement à ce qui se fait dans le domaine concret, environnement et store abstraits ne peuvent plus être composés pour obtenir l'instance abstraite associée à un nom de variable.

*Pile abstraite*

Nous avons choisi une abstraction originale de la pile. Celle-ci réponds à un impératif précis : La taille de la pile abstraite doit être bornée. Or une simple transposition de la pile concrète ne garanti pas cette propriété, car si l'interprétation abstraite doit envisager toute les exécutions possible, des situations de récursivité à l'infini pourraient survenir. Cela aurait pour conséquence la nécessité d'une pile infinie.

Si ce concept peut être envisageable au point de vue théorique, il est impensable dans l'optique d'une implémentation de l'analyse statique. Deux solutions s'offrent donc à nous : adapter le mécanisme régissant la pile abstraite ou se défaire de cette structure, et la simuler autrement. Nous avons choisi la première solution avec le raisonnement ce dessous :

Au niveau concret, la pile permet d'une part de « restaurer » l'environnement « appelant » au terme de l'exécution d'une méthode appelée, et d'autre part d'affecter le résultat de cette exécution à une variable déterminée. Nous allons tenter une approche différente de la pile abstraite permettant de réaliser ces deux objectifs sans recourir à une structure de donnée de taille infinie.

Nous partons du constat suivant : comme toutes les déclarations de variables se font au début de chaque méthode, le domaine de l'environnement (concret) est identique pour l'ensemble des appels de méthode se trouvant dans le corps d'une méthode donnée, mais les environnements sont différents puisque les locations associées aux éléments de ce domaine seront fort probablement différentes.

Si nous nous plaçons dans le cas abstrait, il ne s'agit plus de locations mais bien de types abstraits. Bien sur, ces types peuvent être différents d'un appel à l'autre en vertu des mécanismes de *dynamic binding* qu'il serait malvenu d'ignorer dans notre abstraction. Nous devons toutefois appliquer la logique de borne supérieure présentée supra.

Ainsi, au terme de l'exécution d'une méthode, chaque élément du domaine d'une environnement abstrait donné sera associé au type abstrait représentant une borne supérieure de tous les types abstraits qui auront été pris par la variable au cours de toutes les exécutions possibles de cette méthode.

Cette « borne supérieure de l'environnement » est unique pour chaque méthode. De plus, elle peut être considérée comme une abstraction de l'environnement de la méthode. En outre, comme il y a un nombre fini de méthodes, une pile construite avec une « couche » par méthode serait également bornée.

Toutefois, à côté de l'information de restauration de l'environnement, la pile doit également permettre d'affecter le résultat de la méthode appelée à une variable ou un champ. Cette objectif est assuré en ajoutant à chaque « couche » un ensemble d'informations sur l'affectation du résultat de chaque méthode appelée. Il faut s'assurer que cet ensemble ne sera pas lui même infini, auquel cas notre solution devrait être rejetée. Cette contrainte est satisfaite du fait que le nombre de variables et de champs est lui aussi fini.

*Correspondance entre plie concrète et pile abstraite*

La pile abstraite que nous venons de décrire aura donc une taille bornée. Le prix à payer pour cette propriété est une perte de séquencement de la pile. En effet, dans le cas de la pile concrète, les couches ont été ajoutées successivement au grès des imbrications d'appels de méthodes : l'information se trouvant au sommet correspond à l'information sauvegardée avant l'appel de la méthode en cours d'exécution, la couche qui se trouve immédiatement en dessous a été ajoutée lors de l'appel précédent, etc.

Malheureusement, notre pile abstraite ne permet pas de récupérer de manière précise l'information déterminant le point de programme suivant l'instruction d'appel de méthode, et donc nous ne savons pas précisément cibler l'information à restaurer.

C'est fâcheux, car nous allons devoir essayer de retrouver cette information, ou du moins un ensemble d'informations contenant avec certitude la bonne solution.

Deux pistes peuvent être explorées pour la restauration de l'environnement :

- D'une part tenter toutes les possibilités de restauration de chaque environnement pour être sur d'au moins restaurer celui ou ceux correspondant bien au programme. La seule information utile qui nous permettrait d'effectuer un tri entre les restaurations possibles et celles qui ne conviennent pas est le type statique du paramètre de retour de la méthode qui se termine. Nous pourrons donc éliminer les éléments de la pile dont l'information sur la variable réceptrice du résultat de la méthode n'est pas compatible avec ce type.

- D'autre part récupérer le nom de la méthode en cours puis explorer l'arbre syntaxique afin d'isoler l'ensemble des appels à cette méthode. Sur base de cet ensemble, et plus particulièrement des points de programmes terminaux de ces instructions, parcourir la pile abstraite et dépiler les informations ou le point de retour correspond à l'un des points terminaux de l'ensemble des appels construit précédemment.

  Soient les transitions suivantes, correspondant à des fragments d'un même programme :

  (...)

  $\{p_1\}$ affect $var_1$ $var_2$ $\{p_2\}$

  $\{p_2\}$ ***proc*** $var_3$ $methode_3$ $var_1$ $\{p_3\}$

  (...)

  $\{p_{41}\}$ ***proc*** $var_{20}$ $methode_3$ $var_{21}$ $\{p_{42}\}$

  (...)

  $\{p_{13}\}$ ***proc*** $var_9$ $methode_5$ $var_5$ $\{p_{15}\}$

  $\{p_{14}\}$ ***proc*** $var_5$ $methode_3$ $var_5$ $\{p_{13}\}$

  (...)

  Si la pile abstraite contient les informations suivantes :

| Nom de la méthode | Environnement abstrait | Informations de retour | Nom de la Classe |
|---|---|---|---|
| $methode_2$ | $aenv_2$ | $p_{10}$, **var**<br>$p_{17}$, **var**<br>$p_{50}$, **champ** | $classe_2$ |
| $methode_5$ | $aenv_5$ | $p_{19}$, **var** | $classe_2$ |
| $methode_7$ | $aenv_7$ | $p_{24}$, champ<br>$p_{13}$, **var** | $classe_4$ |
| $methode_8$ | $aenv_8$ | $p_{33}$, champ<br>$p_{42}$, **var** | $classe_4$ |

  Lorsqu'une exécution de la $methode_3$ se termine, on détermine l'ensemble des transitions qui y ont fait appel, et plus particulièrement les points de programmes terminaux de celles-ci $\{p_3, p_{42}, p_{13}\}$. On dépilera uniquement les éléments de la pile qui font référence à ces points de retour (informations grisées) et on générera uniquement les états correspondant à ces informations.

Du système de pile abstraite décrit ci dessus, il faut donc s'attendre à ce que la transition de fin de méthode provoque la création de nombreux états. Cette

multiplication reste toutefois limitée puisque les algorithmes d'interprétation abstraite évitent la mise en attente d'états identiques à ceux qui ont déjà été traités ou mis en attente.

La pile abstraite a la forme suivante :

APile : INomMethode → (AEnv X ***P***(IPtsProg X TypeAppel) X INomClasse) + {***undef***}

Ce qui signifie que pour chaque méthode (appelante) , la pile contiendra un triplet composé de son environnement, d'une liste d'informations sur chaque variable recevant le résultat de l'exécution d'une méthode et sur le point de programme suivant cette exécution. Le troisième élément du triplet est une information sur la classe à laquelle appartient la méthode. En effet, si les noms de méthodes sont à priori uniques, ces méthodes peuvent toutefois être redéfinies au sein de classes dérivées, et le nom de la méthode ne suffit donc plus à l'identifier de manière univoque.

*Etat abstrait*

Signalons tout d'abord que nous venons de définir une sémantique abstraite générique, en ce sens qu'aucune définition du type abstrait n'est jusqu'à présent nécessaire. Nous avons simplement contraint ce dernier à être exprimé en terme d'éléments de l'ensemble des types abstraits, et à permettre le calcul de la borne supérieure de deux de ses éléments.

L'état abstrait sera donc de la forme

Etat abstrait : $< p, P_a, (e_a, s_a) >$

Pour rappel, l'état concret était défini de la façon suivante :

Etat concret : $< p, P, (e, s) >$

### 2.3.2.2. Transitions abstraites

Nous étudierons ici le détail de la transition de retour en fin de procédure, et son effet sur l'état abstrait.

Comme nous l'avons vu dans la sémantique concrète, d'une part cette transition restaure l'environnement de la méthode appelante, ensuite elle affecte le résultat produit par cette méthode à une expression, et enfin elle supprime les informations inutiles de la pile. Nous avons également défini deux pistes pour l'implémentation de cette transition.

*Première piste*

La première piste consistait à balayer l'ensemble des méthodes de la pile abstraite, et pour chacune, l'ensemble des variables et champs de retour ainsi que les points de programme qui y sont associés. Il est clair que cette multiplicité n'est pas optimale, mais en l'état, il nous faut être sûr de générer des états abstraits
dont la concrétisation contient l'état concret correspondant.

Comme dans le cas d'une affectation, le résultat de l'exécution de la méthode qui se termine doit être affecté à l'expression réceptrice (une variable ou un champ). Il s'agit donc de calculer la borne supérieure du type abstrait actuellement associé dans l'environnement (si l'expression de retour est une variable) ou dans l'instance courante (s'il s'agit d'un champ) à l'expression réceptrice et du type abstrait retourné par la méthode.

Nous donnons ici la formalisation de la transition pour une variable comme expression de retour :

$$< p, P_a, (e_a, s_a) > \longrightarrow < q, P_a', (e_a'', s_a) >$$

où { p } ***return*** v

$$<e_a',q> \in \{<env,pts> \mid \exists\, m \in INomMethode,\ t \in INomClasse$$

$$tq\ P_a\ (m) = <env,list,t>$$

$$\Lambda\ \exists\ expr\_ret \in IDes \mid < pts, (\textbf{\textit{var}}, expr\_ret) > \in list$$

$$\Lambda\ SousType\ (TypeAbs(e_a,s_a,v), TypeAbs(e_a', s_a, expr\_ret)\ \}$$

$$e_a'' = e_a'[expr\_ret/\ UnionAbs(TypeAbs\ (e''_a, s_a, expr\_ret), TypeAbs\ (e_a, s_a, v))]$$

La transition a tout d'abord pour but de restaurer un environnement **env.** Nous avons vu que cela n'est pas déterministe, et la transition produira donc un ensemble d'états $< q, P_a', (e_a'', s_a) >$.

Pour chacun, $\mathbf{e_a}$'' sera calculé sur base d'environnements $\mathbf{e_a}$' (par une mise à jour de l'information associée à l'expression de retour) tels qu'ils sont conservés dans la pile $\mathbf{P_a}$. Celle-ci associe à des méthodes **m** des triplets **<env,(point,expr)*,type>** ou **env** est l'environnement de la méthode, **type** fait référence à la classe à laquelle la méthode appartient, et **(point,expr)*** est une liste d'informations sur les différents points de retour consécutifs aux appels se trouvant dans le corps de la méthode **m**.

L'assertion 'SousType(TypeAbs($e_a$,$s_a$,v),TypeAbs($e_a$',$s_a$,expr_ret)' restreint l'ensemble des états à ceux basés sur le dépilement d'une variable de retour compatible avec le résultat de la méthode. Ce dépilement est défini comme suit :

$$P_a' = \begin{cases} P_a\,[m \,/\, \textit{undef}] & \text{si } P_a(m) = < e_a, \{< q, tap >\}, t > \\ P_a\,[m \,/\, < e_a, \{< p_1, tap_1 >, \ldots < p_n, tap_n >\}, t >] & \\ \qquad \text{si } P_a(m) = < e_a, \{<p_1, tap_1>, \ldots <p_n, tap_n>, <q,(\textit{var},v\_ret)>\}, t > & \end{cases}$$

C'est à dire que deux alternatives sont possibles : d'une part le point de programme et la variable de retour dépilés constituaient le seul élément de la liste du triplet **<env,(point,expr)*,type>**. Dans ce cas, l'élément de pile m → **<env,(point,expr)*,type>** n'est plus associé à aucune information de retour au terme de la transition, et il n'est plus utile de le maintenir dans la pile.

D'autre part, les informations dépilées étaient associées à une méthode qui en comportait plus d'une, et il faut se contenter de supprimer ces informations de la liste d'informations de retour, en laissant dans la pile l'information liée aux autres possibilités de dépilement de cette méthode.

*Seconde piste*

Cette seconde piste, plus efficace, consistait à cibler plus précisément les états à générer en ciblant dans l'ensemble du programme les appels à la méthode qui se termine. Ces appels sont de la forme

$\{ p \}$ ***proc*** expr_ret v m $v_1, v_2, ..., v_n$ $\{ q \}$

et l'exécution de la transition abstraite qui leur est associée place **q** dans la pile de la manière suivante :

$m \rightarrow <env,(<p_1,expr_1>, ..., <q,expr_i>, ..., <p_n,expr_n>),type>$

Il vient que l'on peut cibler les environnements à restaurer en balayant l'ensemble des méthodes de la pile abstraite, et pour chacune, l'ensemble des expressions et points de retour, et en ne retenant que les environnements associés à des points de programme du type de **q**, c'est à dire des points que l'on retrouve au terme d'un appel à la méthode qui se termine.

La façon dont sont modifiés l'environnement et la pile sont identiques à la piste précédente, seule le choix des environnements à restaurer change :

$< p, P_a, (e_a, s_a) > \longrightarrow \{ < q, P_a', (e_a'', s_a) > \}$

où $<\{ p \}$ ***return*** $v > = i_1$

$\exists\, dm_1 \in$ **D**eclMeth, $dc_1 \in$ **D**efClasse : $i_1 \in$ **I**nstr $\cap\, dm_1$ , $dm_1 \in dc_1$

$\forall dm_2 \in$ **D**eclMeth, $\forall\, i_2 \in$ **I**nstr $\cap\, dm_2$ , $p_2 \in$ **P**tsProg ; $expr_1, expr_2, expr_i \in$ **E**xpr

nmeth $\in$ **N**omMethode

$i_2 = <p_2$ ***proc*** $expr_1\ expr_2$ nmeth $(expr_i)^*\ q>$

$\Lambda$ nmeth = $Nom(dm_1)$

$\Lambda$ $Type(expr2) = Nom(dc_1) = t$

$\Lambda$ $q \in proj_1(proj_2(P_a\ (nmeth))$

$\Lambda$ $P_a\ (nmeth) = <e_a',list,t>$

$\Lambda\ \exists$ expr_ret $\in$ **D**es $| < q, (\textit{var}, expr\_ret) > \in$ list

Si d'une part $\mathbf{i_1}$ est la transition courante, et qu'elle constitue l'ultime instruction d'une méthode $\mathbf{dm_1}$, elle-même déclarée dans une classe $\mathbf{dc_1}$,

Si, d'autre part, on a l'ensemble des instructions $\mathbf{i_2}$ déclarées dans des méthodes $\mathbf{dm_2}$ implémentant des appels à la méthode $\mathbf{dm_1}$ de la classe $\mathbf{dc_1}$ (formalisé par la condition nmeth = $Nom(dm_1)$ )

Si, en plus, pour cette instruction $\mathbf{i_2}$, il y a dans la pile une information de restauration au point de programme **q** atteint au terme de l'exécution de $\mathbf{i_2}$ ( décrit par la condition $\mathbf{P_a\ (nmeth) = <e_a',list,t>}$)

Alors l'expression **expr_ret** peut être affectée du résultat de la méthode, l'environnement $\mathbf{e_a'}$ restauré (moyennant la mise à jour de $\mathbf{e_a'}$) et le point **q** devenir le point de programme courant.

### 2.3.3. Choix d'un domaine abstrait particulier...

Nous avons choisi le domaine abstrait de base. Dans cette optique, la fonction d'abstraction sera l'identité.

Cependant, la fonction de concrétisation associera un type abstrait *atype* à un ensemble types concrets représentant toutes les classes concrètes qui en sont dérivées. Ce qui donne formellement

$$\mathbb{A}\text{Type} = \mathbb{I}\text{NomClasse} \cup \{\textbf{\textit{unknown}}\}$$

$$Cc : \mathbb{A}\text{Type} \rightarrow P(\mathbb{I}\text{NomClasse} \cup \{\textbf{\textit{unknown}}\})$$

atype ~> {T' | T' ≤ atype}
ou chaque type T' est une spécialisation du atype

bottom ~> ***unknown***

La concrétisation d'un type abstrait est donc un arbre de types concrets dont la racine donne le type abstrait. Le type abstrait des expressions non typées (***unknown*** associé à la valeur null) est bottom.

En outre, le calcul d'une borne supérieure à deux types abstrait se fait en recherchant le plus précis des supertypes (concret) communs à tous les éléments des types abstrait dont on souhaite obtenir la borne supérieure. Si aucune solution n'est trouvée, la borne supérieure sera $\mathbb{T}$ (« *top* », l'élément abstrait porteur de l'information la moins spécialisée). Sinon, ce sera l'abstraction de ce supertype commun.

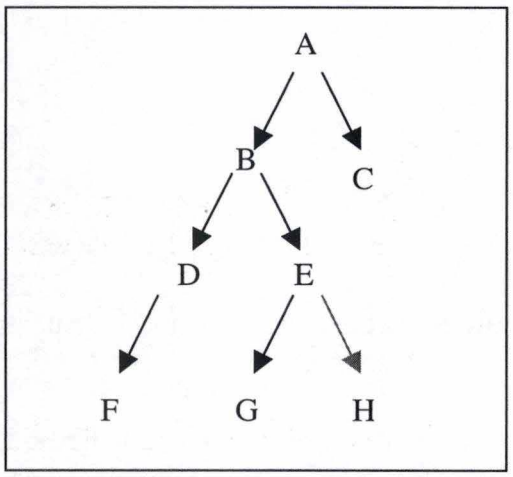


*Figure 2.1 : types abstraits et hiérarchie de types*

INomCLasse = {A, B, C, D, E, F, G, H} + ***undef***

AType = {A (= *top*), B, C, D, E, F, G, H}

Abs (F) = {F}

Cc (E) = {E, G, H}

UnionAbs(E, F) = {B}

## 2.3.4. Preuve de correction

La construction des preuves de correction suit un schéma classique en interprétation abstraite : On sait que l'exécution concrète est vue comme une suite de transitions $t_i$ faisant successivement passer le système d'un état $c_i$ à un état $c_j$. On dispose également de fonctions réciproques d'abstraction d'états concrets et de concrétisation d'états abstraits.

Le principe est de prouver, sou l'hypothèse que l'état concret avant l'exécution de la transition était bien dans la concrétisation de l'état abstrait correspondant, qu'alors l'état concret après la transition sera également dans la concrétisation de l'état obtenu en exécutant la transition abstraite sur l'état abstrait mentionné ci dessus. En d'autres termes :

Hypothèse : $c_i \in C_C(a_i)$

Thèse : $c_j = trans_c(c_i) \in C_C(a_j)$ si $a_j = trans_a(a_i)$

### 2.3.4.1. Preuve de correction de la transitions abstraites de fin de méthode (return)

Nous décrirons ci dessous la preuve de correction de la transition que nous avons étudiée précédemment : l'instruction de fin de méthode.

{ p } ***return*** v

Comme nous l'avons dit, nous devons prouver que l'état concret modifié par la transition fait partie de la concrétisation de l'état abstrait modifié par la transition abstraite correspondante. L'état concret au terme de la transition est l'état restauré sur base des informations qui se trouvent au sommet de la pile, mais ou l'environnement est modifié pour y faire apparaître l'affectation du résultat de la méthode à la variable de retour, ce qui donne

$c' = < q, P', (e'', s) >$

où $\exists$ e', expr_ret | $P = <e', q, (\textbf{\textit{var}}, expr\_ret)> : : P'$

$\Lambda \quad e'' = e'[expr\_ret / Loc(e, s, v)]$

La façon dont nous avons défini l'état abstrait après la transition de fin de méthode a été vu au point précédent. Pour rappel, il s'agit en fait d'un ensemble d'états abstraits définis comme suit :

$a' = < r, P_a', (e_a'', s_a) >$

où $\exists\, e_a' : < e_a', r > \in \{ <env, pts> \mid \exists$ m, t tq $P_a(m) = < env, list, t >$

$\Lambda \quad \exists$ v_ret' $\in$ IDes $\mid < pts, (\textbf{\textit{var}}, v\_ret') > \in$ list

$\Lambda \quad SousType(TypeAbs(env, s_a, v), TypeAbs(e_a', s_a, v\_ret')) \}$

$\Lambda \quad ta_1 = TypeAbs(e''_a, s_a, v\_ret)$

$\Lambda \quad ta_2 = TypeAbs(e_a, s_a, vt)$

$\Lambda \quad e_a'' = e_a' [v\_ret / UnionAbs(ta_1, ta_2)]$

$\Lambda$ $\begin{cases} P_a' = P_a[m / \textbf{\textit{undef}}] \\ \quad \text{si } P_a(m) = < e_a, \{< q, tap >\}, t > \\ P_a[m / proj_2(P_a(m)) \setminus \{<q, (\textbf{\textit{var}}, v\_ret')>\}, t] \\ \quad \text{sinon} \end{cases}$

}

L'étape suivante consiste à prouver que les différents éléments de l'état concret appartiennent bien à la concrétisation de l'état abstrait.

- La première composante de l'état est le point de programme de retour. Il faut prouver que le point de retour concret **q** fait bien partie de la concrétisation des points de programmes de l'état abstrait.

  La transition de fin de méthode qui nous occupe est nécessairement couplée, au niveau concret, à un appel de méthode qui lui correspond et ou une nouvelle couche a été déposée sur la pile avec comme point de retour **q**.

  Or, nous savons, par la définition de la transition abstraite d'appel de méthode, que **q** a également été empilé dans la pile abstraite, même si à posteriori nous ne savons pas dire où. Comme la transition abstraite de fin de méthode va activer un nouvel état pour chaque information empilée lors d'un appel de méthode, on est sur que le point de programme **q** fera partie d'un des états créés, ce qui valide l'hypothèse. La restriction sur le fait que la transition abstraite de fin de méthode ne créé que des états pour les dépilements ou la variable de retour est « compatible » avec le résultat de la méthode n'hypothèque pas ce fait puisqu'au niveau concret, le programme est syntaxiquement correct, ce qui est une condition suffisante. Nous pouvons formaliser le fait que **q** appartienne à {r} défini ans la transition abstraite ci dessus par

  $$\exists\ m, env_a, l : P_a(m) = < env_a, \{<q, (\textbf{\textit{var}}, v\_ret) >\} \cup l, t >$$

- L'environnement, ou plutôt le couple < environnement, store > concret fera partie de la concrétisation de l'environnement abstrait d'une part si pour tout élément du domaine de l'environnement concret, il existe un élément identique dans le domaine de l'environnement abstrait. D'autre part, le type associé à cet élément par l'environnement concret appartient à la concrétisation du type abstrait associé à l'élément de même nom de l'environnement abstrait. En d'autre termes

  $$\forall\ v_i \in dom(e') : Type(e', s, v_i) \in C_C(e_a'(v_i))$$

  Le point précédent nous a assuré que l'un des états généré suite à la transition de fin de méthode contenait le point de programme correspondant au point de programme **r** présent dans l'état concret activé au terme de la transition de fin de méthode.

  Si nous prenons la méthode à laquelle ce point de programme se trouvait associé au sein de la pile abstraite, son environnement a été empilé lors de la transition d'appel de la méthode qui se termine par la transition qui nous occupe. Si la variable de retour est un champ de l'instance courante, l'environnement qui sera restauré sera celui qui fut empilé. Comme l'environnement concret empilé lors de l'appel de la méthode était, par hypothèse, inclus dans la concrétisation de l'environnement abstrait, la thèse est démontrée.

  Il nous reste le cas ou l'expression de retour est une variable. Au moment de l'empilement de l'environnement, lors de l'appel de la méthode qui se termine par la transition courante, l'environnement empilé appartenait bien à la concrétisation de l'environnement abstrait associé. Ceci signifie que pour chaque élément du domaine de cet environnement concret, le type de l'instance qui lui était associé par la

composition de l'environnement et du store faisait bien partie de la concrétisation du type abstrait associé à ce même élément, mais dans l'environnement abstrait.

Dans le cas qui nous occupe, seule la variable de retour sera modifiée au concret qu'à l'abstrait. Nous pouvons donc réduire notre problème à prouver qu'après l'exécution de la transition de fin de méthode, le type de l'instance associé à la variable de retour par la composition de l'environnement et du store concrets font bien partie de la concrétisation du type abstrait associé à la variable de retour.

Or nous savons que le type abstrait qui sera associé à la variable de retour sera la borne supérieure du type abstrait qu'elle possédait initialement et du type abstrait de la valeur retournée par la fonction. Et par définition, cette borne supérieure contient au moins l'abstraction du type concret qui sera associé à la variable de retour dans l'environnement et le store concret. En d'autres termes,

$(e', s) \in C_C(e_a')$

$\Leftrightarrow \quad \forall\, v_i \in dom(e') : Type(e', s, v_i) \in C_C(e_a'(v_i))$

$\Leftrightarrow \quad \forall\, v_i \neq v_1 \quad : Type(e', s, v_i) \in C_C(e_a'(v_i))$

$\qquad\qquad \Lambda \quad Type\ (e', s, v_1) \in C_C\ (e_a'(v_1))$

$\Leftrightarrow \quad \forall\, v_i \neq v \quad : Type(e, s, v_i) \in C_C(e_a(v_i))$ ce qui est vrai par hypothsèe

$\qquad\qquad \Lambda \quad Type\ (e, s, v_2) \in C_C\ (e_a'(v_1))$ ce qui est vrai si

$Type(e,s,v_2) \in \quad TypeAbs(e_a,s_a,v_2)$

$\qquad\qquad \subseteq \quad UnionAbs(TypeAbs(e_a,s_a,v_1), TypeAbs(e_a,s_a,v_2))$

- Le store quand à lui n'est pas modifié par la transition de fin de méthode si le résultat de celle-ci est affecté à une variable. Ceci est valable tant au niveau concret qu'au niveau abstrait.

  Par contre, si le résultat de la méthode est affecté à un champ de l'instance courante, nous savons que l'élément du domaine du store abstrait correspondant au type de ce champ sera remplacé par la borne supérieure du type abstrait auquel il était associé précédemment et du type abstrait du résultat de la méthode.

$$s_a' = s_a[t \,/\, (s_a\ t)[v\_ret\ UnionAbs(TypeAbs(i_a,s_a,v\_ret), TypeAbs(e_a,s_a,v))])]$$

Il reste à savoir si la concrétisation de ce store abstrait modifié couvre bien le store concret après la transition. Ce dernier est modifié comme suit :

$$s' = s\ [e(this)/(proj_1(s(e(this))),\ proj_2(s(e(this)))[v\_ret/Loc(e,s,v)]]$$

On constate que le type du champs affecté devient celui du résultat de l'exécution de la méthode, or c'est précisément l'abstraction de ce résultat qui est ajouté, via borne supérieure, au store abstrait.

- Concernant la pile, la définition de la pile abstraite nous apprend qu'au terme de la transition de fin de méthode, chacun des états générés contient une pile ou l'information sur le point de retour auquel devra se poursuivre l'exécution a été supprimé, et, si la méthode correspondant au sein de la pile ne possède plus d'éléments dans sa liste de points de retour, la couche associée est également enlevée.

  Si nous nous plaçons dans le cas ou la pile concrète faisait bien partie de la concrétisation de la pile abstraite avant l'appel de la méthode qui se termine, alors il y a au moins un état généré qui restaurera cette pile telle quelle était avant l'appel. Dans ce cas, la pile concrète sera bien présente dans la concrétisation de la pile abstraite.

## 2.4 Implémentation avec l'algorithme multivariant

Nous décrivons dans ce point l'implémentation de la méthode présentée ci-dessus. Nous avons choisi d'implémenter celle-ci en utilisant le langage Java lui-même. Ce choix peut paraître curieux dans la mesure ou un langage de programmation fonctionnel se prête mieux à l'implémentation de ce genre de problèmes. Toutefois, la perspective d'adaptation de cette analyse en vue de l'intégrer dans le projet (lui même massivement développé à l'aide de Java) nous a fait pencher pour ce langage.

### 2.4.1. Architecture de l'analyseur

Nous l'avons dit, l'analyse statique est souvent utilisée lors de la compilation. Ainsi, notre analyseur travaillera-t-il au niveau sémantique.

Toutefois, afin de le rendre relativement autonome, nous avons prévu de faire précéder la phase d'analyse proprement dite d'un module rustique de chargement du programme en mémoire.

Notre application se divise donc en deux phases : d'une part un module de chargement dont le but est de construire l'arbre syntaxique, d'autre part l'analyseur proprement dit, implémentant l'algorithme multivariant.

Le schéma suivant donne l'architecture générale de l'analyseur. Chaque module sera décrit par la suite.

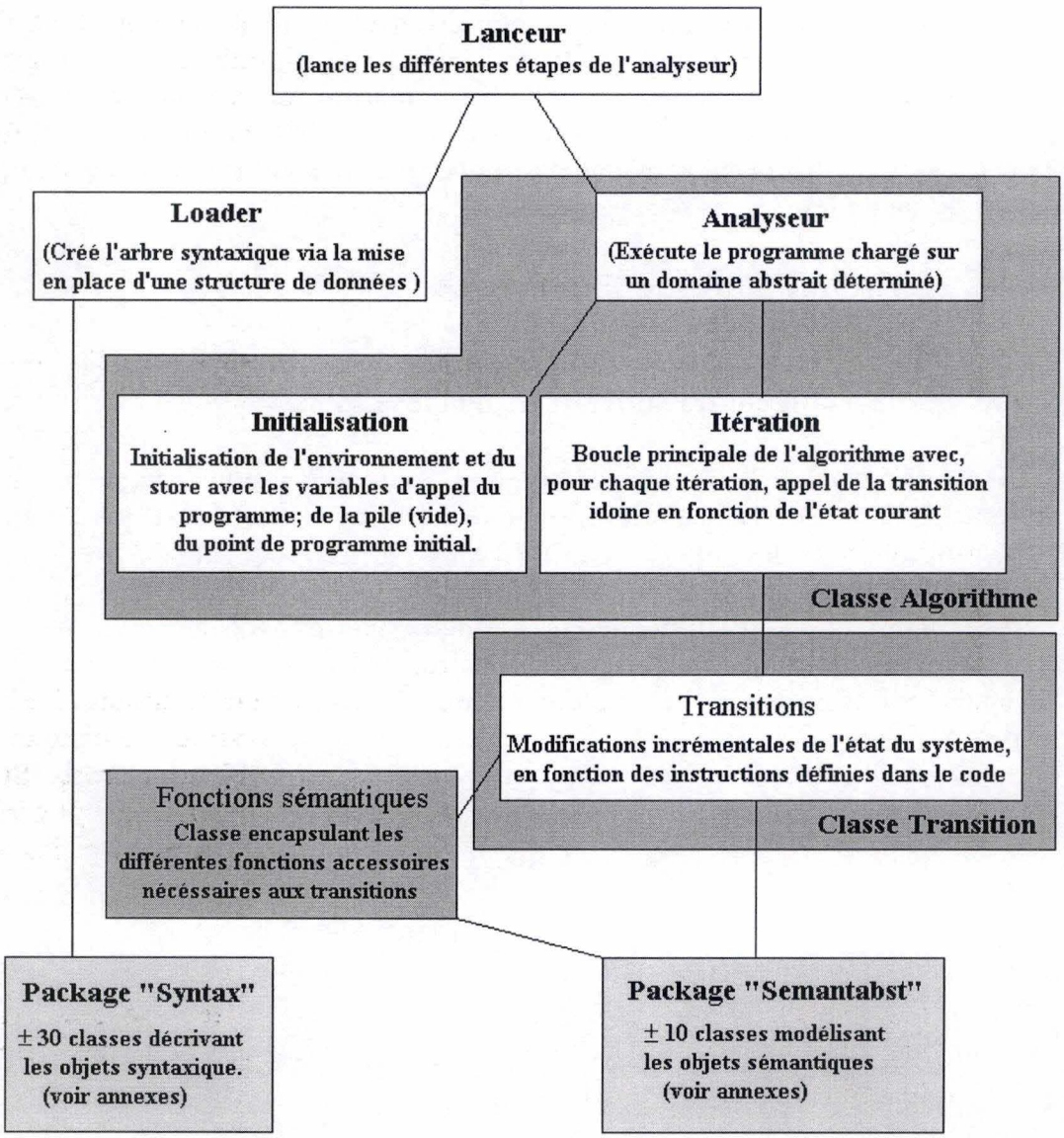


*Figure 2.2 : architecture de l'analyseur*

#### 2.4.1.1. Loader

Le but de ce module est de créer l'arbre syntaxique correspondant au code d'un programme qu'il prends en entrée. Nous avons choisi d'implémenter ce module sous la forme d'un générateur d'objet. En effet, prévoir un parseur complet dépassait le cadre de ce travail. Nous avions plutôt besoin d'un traducteur simple. Dans cette optique, nous avons défini un pseudo langage rustique, à mi chemin entre le sous langage de Java que nous utilisons et la structure de l'arbre syntaxique, et aisément traduisible en une seule passe. Le pseudo langage est défini à l'annexe 1.

La sortie produite par le module est le code source d'un objet Java implémentant une méthode Load() retournant un objet syntaxique de classe **Prog**, en fait la racine d'un arbre syntaxique complet. L'exécution de la tâche assignée au module se déroule donc en trois temps :

- La transcription du programme à analyser en pseudo langage
- L'exécution du générateur sur le pseudo code rédigé à l'étape précédente
- La compilation de la source créée par la seconde étape

Le générateur en lui même ne présente pas de difficulté particulière. Il se contente de convertir le pseudo code en objets syntaxiques. Pour cela, chaque ligne du pseudo code est définie pour correspondre à un nœud de l'arbre. Surtout, le pseudo code contient la totalité de l'information nécessaire, y compris les mécanismes de labélisation et les données relatives aux liens entre les nœuds de l'arbre.

#### 2.4.1.2. Package syntaxique

Il s'agit d'un ensemble de classes modélisant les différents objets syntaxiques définis par la syntaxe abstraite du sous langage de Java sur lequel nous travaillons.

En outre, comme ces classes sont susceptibles d'être modifiées et intégrées au projet global, des mécanismes de vérification de correction en cascade sont implémentés pour chaque classe. Ainsi, l'appel de la méthode de vérification de correction sur le nœud racine déclenche le processus de vérification à travers l'ensemble des nœuds du graphe.

Nous donnons ci-dessous l'arbre des classes qui composent ce package. Les relations en claire représentent l'implémentation d'interfaces alors que les relations foncées représentent des héritages. Les termes entourés représentent des instances remarquables (statiques) alors que les noms qui ne sont pas entourés représentent des classes.

Notons que la table à accès direct **DefClasseSet** permet d'accéder directement aux objets modélisant les déclarations de classes sur base de leur nom.

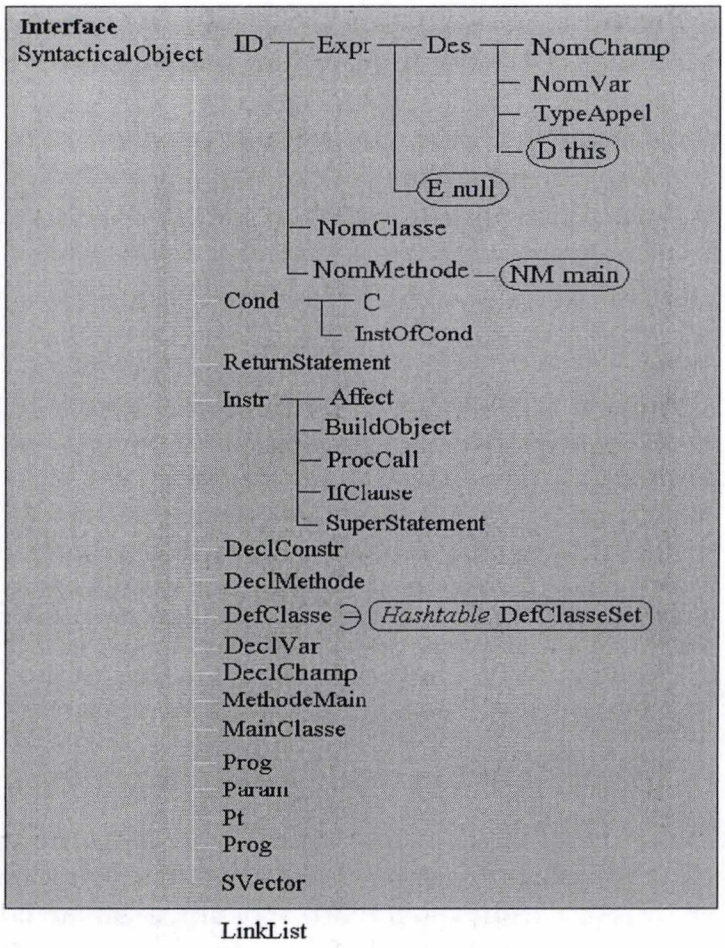


*Figure 2.3 : hiérarchie des classes modélisant la syntaxe abstraite*

### 2.4.1.3. Analyseur

Ce module implémente l'algorithme multivariant et est divisé en deux parties d'une part l'initialisation du premier état abstrait et d'autre part un processus itératif traitant chaque état de l'ensemble des états actifs.

L'initialisation créé donc un état composé d'une pile vide, d'un environnement garni avec les paramètres d'exécution du programme, d'un store contenant uniquement les types abstraits liés à ces paramètres et d'un point de programme correspondant à la première instruction de la méthode main. En outre, l'ensemble des états actifs est créé avec ce seul élément, alors que l'ensemble des états traités est initialisé vide.

Le processus itératif est relativement simple. Il prélève un état de l'ensemble S, l'ajoute à l'ensemble R et fait appel à la transition idoine. Les transitions sont regroupées au sein d'une classe distincte qui sera détaillée ultérieurement.

Nous attirons l'attention du lecteur sur un aspect spécifique de notre implémentation: nous disposons en l'état d'un programme à analyser sous forme

d'arbre syntaxique et nous envisageons l'exécution abstraite sans, semble-t-il, être passé par une étape de création des structures abstraites. En fait, cette abstraction ne fait pas l'objet d'une étape à part entière, mais bien d'une application au fur et à mesure, en ce sens que les nouveaux états abstraits sont établis sur base des états précédents et des instructions labélisées, la mise en œuvre de la transition abstraites correspondante étant déduite de la transition concrète.

#### 2.4.1.4. Transitions

C'est à ce niveau que l'exécution abstraite a effectivement lieu. En effet, pour chaque transition présentée au point 2.3.2, le ou les états abstraits terminaux sont générés conformément à la sémantique abstraite.

Concrètement, nous avons implémenté une méthode par transition. Chacune d'entre elles présentant les même paramétrages : un état initial en entrée et un ensemble d'états en sortie. Le module dispose en outre d'un package de classes utiles à la sémantique abstraite contenant

- Un ensemble de fonctions sémantiques (voir [HHN2000])
- Un ensemble de classes modélisant les concepts de la sémantique abstraite. Nous donnons ci-dessous l'arbre des classes qui composent ce package.

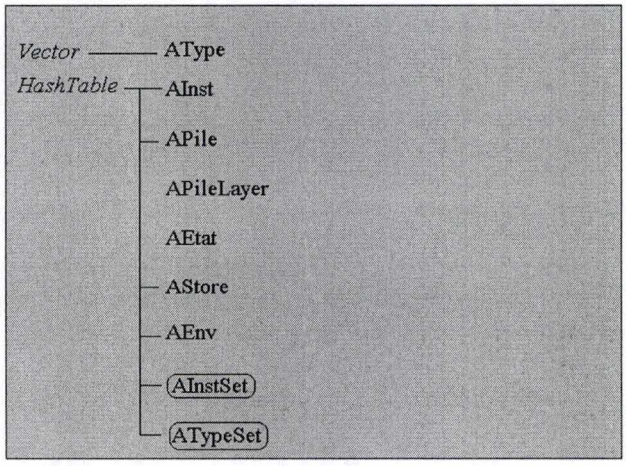


*Figure 2.4 : hiérarchie des classes modélisant la sémantique*

Signalons que si l'implémentation des transitions en elles-mêmes n'a pas représenté une difficulté insurmontable, c'est du côté des fonctions sémantiques que les plus épineux problèmes ont surgi. En effet, la spécification de ces fonctions dans le formalisme que nous avons choisi cachait soigneusement (sic) un certain nombre de difficultés incontournables lors de l'implémentation, C'est ainsi que certaines fonctions « magiques », comme par exemple la fonction chargée de retourner le type abstrait de l'objet courant sont loin d'entraîner une implémentation triviale. C'est la principale raison pour laquelle l'ensemble des transitions n'a pu faire l'objet d'une implémentation dans les temps impartis à la réalisation de ce mémoire, ce qui a sérieusement hypothéqué nos chances d'obtenir et d'exploiter les résultats de notre étude de cas.

# Chapitre 3

## Un aspect particulier : l'affichage de l'environnement et du store

Pour éviter toute confusion, nous rappelons au lecteur que le chapitre précédent présentait l'analyse statique d'un sous langage plus réduit que celui utilisé généralement au sein du projet. Pour une définition complète de celui-ci, on se rapportera à [POLLET99].

Dans ce chapitre, nous aborderons la mise en œuvre d'un aspect particulier du projet JavabInt. Plus particulièrement, il s'agit d'un module d'affichage d'un couple environnement/store. Le Store est une structure de donnée qui peut être vue comme un graphe dont les sommets représentent des locations, et les arcs des références entre celles-ci. Ce graphe est étendu par l'ajout de l'information contenue dans l'environnement sous forme de sommets spécifiques .

Pour illustrer notre propos, nous utiliserons un exemple concret. Soit le programme suivant :

```
public class Pgm
 {
// Vars
   int      message;
   int      surface;
   Polygon     shape1;
   Triangle    shape2;
// Constructors
   public Pgm(int chx_title)
   {                                                    (1)
     if !( chx _title == 0)
           {message = chx_title; }
      else {message = 321 ; }
     shape1 = new Triangle(60,60,60) ;                  (2)
     shape2 = shape1 ;
     surface = shape2.calculeSurface() ;
     System.out.println("Surface = " + surface) ;
     }
// Methods
public static void main(String[] args)
  {
   private  Pgm      prog ;
   prog = new Pgm(2);
  }
}
```

Avant l'exécution de l'instruction de la ligne 1, la méthode main a fait appel au constructeur de la classe Pgm. L'environnement est constitué des informations suivantes :

| Nom de variable | this | chx_title |
|---|---|---|
| Location | 1 | 2 |

Et le store est défini comme suit :

| Location | Instance | | | |
|---|---|---|---|---|
| 1 | int | Message | Null | |
| | Int | Surface | Null | |
| | Polygon | shape1 | Null | |
| | Triangle | shape2 | Null | |
| 2 | int | | | 2 |
| | | | | |

La représentation graphique associée sera donc

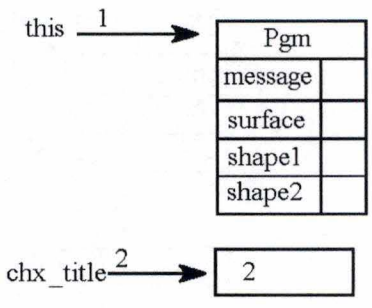


*Figure 3.1 : représentation graphique simple de l'environnement et du store*

Au terme de l'exécution de l'instruction de la ligne 2, les champs **message** et **shape1** auront reçu un contenu. Environnement et store refléteront ces modifications de la manière suivante :

| Nom de variable | this | chx_title |
|---|---|---|
| Location | 1 | 2 |

| Location | Instance | | | | | |
|---|---|---|---|---|---|---|
| | | Type du champ | Nom du champ | Location | Val | Type de l'instance |
| 1 | | String | message | 2 | | Pgm |
| | | Integer | surface | Null | | |
| | | Polygon | shape1 | 7 | | |
| | | Triangle | shape2 | Null | | |
| 2 | | Int | | | 2 | |
| 7 | | int | cote1 | | 60 | Triangle |
| | | int | cote2 | | 60 | |
| | | int | cote3 | | 60 | |
| | | int | cote4 | | 0 | |
| | | int | cote5 | | 0 | |
| | | int | cote6 | | 0 | |
| | | int | hauteur | | 52 | |

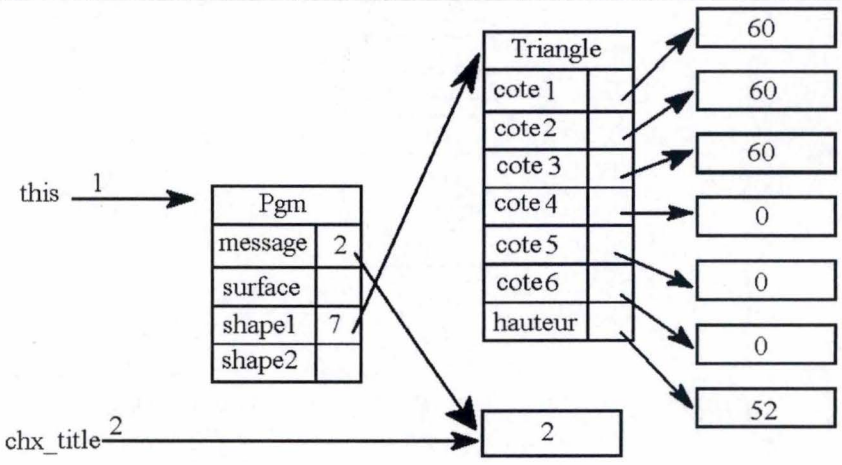


*Figure3.2 : représentation graphique de l'environnement et du store*

## 3.1 Terminologie

### 3.1.1. Vocable de la théorie des graphes

Nous reprécisons ici un ensemble de concepts qui seront utilisés tout au long de ce chapitre. Ces définitions sont largement inspirées de [FICHEFET97]

#### 3.1.1.1. Graphe

Un graphe G est un couple (X,U) composé

- de l'ensemble $X = \{x_1, x_2, \ldots, x_n\}$ dénombrable des sommets du graphe.
- de l'ensemble $U = \{u_1, u_2, \ldots, u_m\}$ des arcs du graphe. Les éléments de U sont également représentés par des paires de sommets $(x_i, x_j)$ constituant les extrémités d'un arc.

$X = \{x_1, x_2, x_3, x_4, x_5, b\ x_6\}$

$U = \{u_1, u_2, u_3, u_4, u_5, u_6, u_7, u_8,\}$

$u_1 = \{x_1, x_2\}$ $u_5 = \{x_5, x_5\}$

. . .

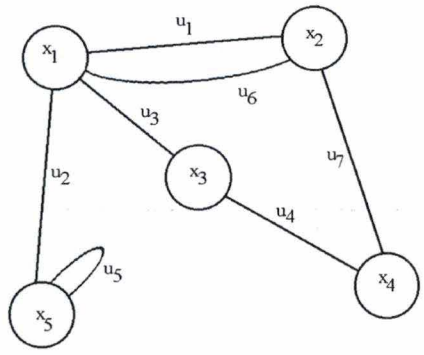


*Figure3.3 : graphe*

#### 3.1.1.2. Graphe orienté

Un graphe orienté G est un graphe dont la définition d'arc est étendue. Cette extension permet de distinguer les extrémités des arcs.

Pour un arc orienté $u_k = (x_i, x_j) \in U$, on dit que le sommet $x_i$ est l'extrémité initiale de l'arc et $x_j$ en est l'extrémité terminale.

$X = \{x_1, x_2, x_3, x_4, x_5, x_6\}$

$u_1 = \{x_2, x_1\}$

$\{x_1, x_2\} \notin U$

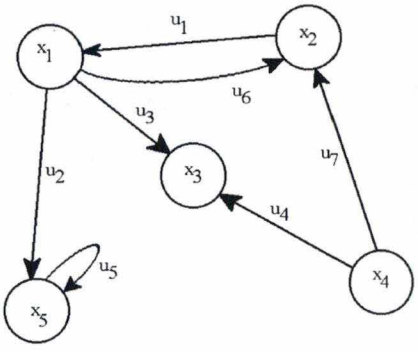


*Figure3.4 : graphe orienté*

#### 3.1.1.3. Graphe simple

Un graphe G est un graphe simple si pour tous sommets $x_i$ et $x_j$, il existe au plus un arc $(x_i, x_j)$ reliant ces sommets. En outre, les arcs ayant leurs deux extrémités sur le même sommet sont également exclus.

Un graphe simple orienté admet, pour toute paire de sommets $x_i$ et $x_j$, $(i \neq j)$ au plus un arc $(x_i, x_j)$ et au plus un arc $(x_j, x_i)$

### 3.1.1.4. Arcs successifs

Deux arcs orientés $u_1 = (x_i , x_j)$ et $u_2 = (x_k , x_l)$ sont successifs si $x_j = x_k$. C'est par exemple le cas pour les arcs $u_1$ et $u_3$ de la figure du point 3.1.1.2.

### 3.1.1.5. Chemin

Une suite d'arcs orientés $\mu = (u_1, u_2, \ldots , u_i, u_{i+1},\ldots, u_m )$ est un chemin si $\forall\, i \in [1, m-1]$, les arcs $u_i$, $u_{i+1}$ sont successifs, c'est à dire si l'extrémité terminale de $u_i$ est égale à l'extrémité initiale de $u_{i+1}$.

La longueur du chemin vaut m, soit le nombre d'arcs qui composent le chemin.

Par extension, l'extrémité initiale (resp. terminale) d'un chemin est l'extrémité initiale (resp. terminale) de son premier (resp. dernier) arc.

### 3.1.1.6. Circuit

Un chemin $\mu = (u_1, u_2, \ldots , u_i, u_{i+1},\ldots, u_m )$ est un circuit si son extrémité terminale coïncide avec son extrémité initiale. On parle également de cycles pour désigner les circuits.

### 3.1.1.7. Successeur

Un sommet $x_j$ est successeur d'un sommet $x_i$ dans le graphe orienté G = (X,U) s'il existe un chemin $\mu$ ayant $x_i$ comme extrémité initiale et $x_j$ comme extrémité terminale.

Par restriction de la définition précédente, un sommet $x_j$ est successeur direct d'un autre sommet $x_i$ s'il en est successeur et s'il existe un chemin de $\mu$ longueur 1 (composé d'un seul arc) ayant $x_i$ comme extrémité initiale et $x_j$ comme extrémité terminale.

### 3.1.1.8. arbre

Un graphe orienté G = (X,U) est un arbre si

$$(\forall\, x_i \in X)\ ,\ (\forall\, x_j \in X)\ ,\ (\forall\, x_k \in X)\ \mid\ x_k \neq x_i\ ,\ (x_i\ ,\ x_j) \in U \Rightarrow\ (x_k\ ,\ x_j) \notin U$$

### 3.1.1.9. Racine

La racine d'un arbre G = ( X , U ) est un sommet qui n'est successeur d'aucun autre sommet de X.

### 3.1.1.10. Feuille

Une feuille d'un arbre G = ( X , U ) est un sommet qui n'a pas de successeur dans X.

$X = \{x_1, x_2, x_3, x_4, x_5, x_6\}$

$U = \{u_1, u_2, u_3, u_4, u_5\}$

G = {X,U} est un arbre

$x_1$ est une racine

$x_5$ est une feuille

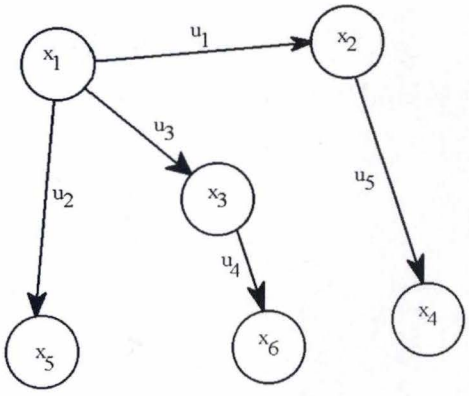


*Figure3.5 : arbre*

Les deux notions précédentes peuvent, abusivement, être utilisées plus généralement pour les graphes orientés.

### 3.1.1.11. Hiérarchie

Une hiérarchie sur un graphe G = (X,U) est une application

$$H : X \rightarrow I = \{1, 2,...,k\} \subset \mathbb{N}$$

associant à tout sommet $x_i \in X$ un entier $i \in I$

Un exemple de hiérarchie intéressante est la hiérarchie de rang sur les graphes orientés sans circuits : Le rang $r_i$ d'un sommet $x_i$ est le maximum des longueurs de chemins d'extrémité terminale $x_i$.

La hiérarchie de rang $H_r$ associe a chaque sommet $x_i$ est un entier $n_i = r_i + 1$ .

$n_i$ est appelé rang de $x_i$.

### 3.1.1.12. Graphe hiérarchisé

Par abus de langage, on appelle Graphe hiérarchisé un graphe sur lequel est établi une hiérarchie de rang.

### 3.1.1.13. Portée d'un arc

Soit $u_i = (x_i , x_j)$, un arc d'un graphe hiérarchisé. La différence des niveaux $n_j - n_i$ est appelée portée de l'arc $u_i$.

Le graphe ci contre est un graphe hiérarchisé.

Le sommet $x_1$ est de rang 0 alors que le sommet $x_7$ est de rang 2.

L'arc $u_4$ a une portée de 2

Par convention, l'orientation des arcs de haut en bas est induite.

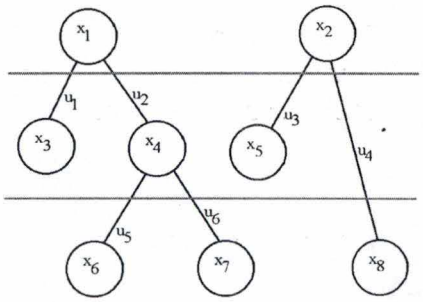


*Figure3.6 : graphe hiérarchisé*

### 3.1.1.14. Graphe acyclique

Un graphe acyclique est un graphe orienté sans cycles.

## 3.2 Affichage d'une représentation optimale de graphe

Le problème du positionnement d'entités en relations dans un espace à deux dimensions n'est pas neuf. L'industrie de l'électronique notamment y est confrontés tant dans la conception des circuits intégrés que dans le design des circuits imprimés. On retrouve également des problématiques semblables dans de nombreuses branches de l'informatique, ou la représentation de concepts en relations est monnaie courante (schémas entités – associations, diagrammes de flux, topologies de réseaux...).

Ceci étant, le positionnement des nœuds et des arcs d'un graphe quelconque est un problème très complexe pour peu que l'on le subordonne à la mise en œuvre de règles de lisibilité. La détermination automatique d'un positionnement optimal fait donc massivement appel à des heuristiques. De nombreuses études sont menées dans ce domaine et donnent des résultats plus qu'encourageants. I y sera fait référence tout au long de ce point.

### 3.2.1. Paradigmes d'affichage de structures de données en graphes

G. Di Battista et al. [DiBattista99] introduisent trois classes de paramètres pour caractériser le choix d'affichage d'un graphe : les conventions graphiques, les facteurs esthétiques et les contraintes.

*Conventions graphiques*

Les conventions graphiques expriment les règles de bases qui doivent toujours être vérifiées par la solution graphique. La plupart du temps, il s'agit des conventions sur la représentation des sommets et des arcs. Les conventions graphiques les plus courantes portent sur l'alignement des sommets, l'absence de brisure ou l'orthogonalité des arcs. L'absence de croisements entre arcs (planéité) est également une convention fréquemment utilisée. Un paradigme d'affichage constitue une combinaison déterminée de conventions graphiques.

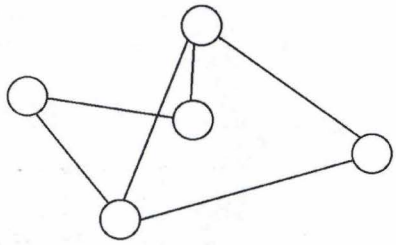

*Figure 3.7 : Arcs en ligne droite*

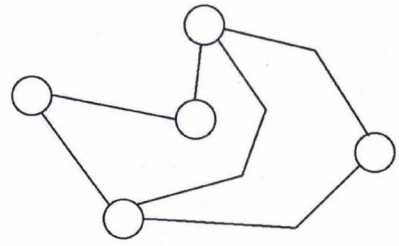

*Figure 3.8 : Arcs en lignes brisées et planéité*

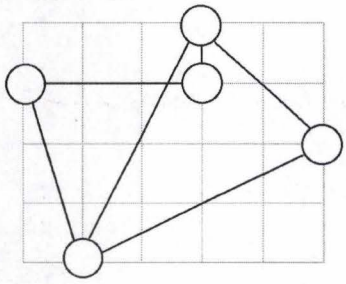

*Figure 3.9 : Sommets disposés sur une grille*

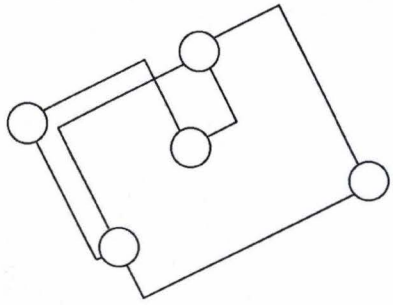

*Figure 3.10 : Arcs orthogonaux*

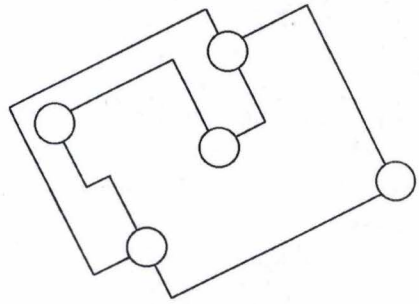

*Figure 3.11 : Arcs orthogonaux et planéité*

*Facteurs esthétiques*

Les facteurs esthétiques concernent des propriétés générales du graphe qu'il convient de respecter dans la mesure du possible car ils améliorent la lisibilité de la solution graphique. En outre, on retrouve parfois dans les facteurs esthétiques des conventions qu'il n'a pas été possible de mettre en œuvre avec un graphe donné, mais qu'il est souhaitable de voir satisfaites dans la mesure du possible. (certains graphes n'admettent par exemple pas de représentation planaire, mais il peut être bon de tendre autant que possible vers la planéité). On retrouve dans cette catégorie l'absence de croisements entre arcs, la réduction des espaces vides, la symétrie dans les graphes orientés ou la minimisation de la longueur

totale des arcs. On notera que certains facteurs esthétiques ont un effet négatif l'un sur l'autre. Il est donc intéressant de classer les facteurs esthétiques en fonction de leur importance.

*Contraintes*

Les contraintes sont des paramètres qui ne s'appliquent qu'à des sous graphes bien définis. Il s'agit par exemple de placer les racines du graphe à l'extérieur, de centrer un sommet particulier, d'aligner ou « agréger » les sommets partageant une même propriété... Ces contraintes peuvent revêtir un caractère obligatoire ou être un idéal vers lequel il faut tendre.

Ceci étant, la lisibilité d'un graphe est un concept largement subjectif. Un graphe sera cerné au premier coup d'œil par un lecteur mais pourra paraître confus et difficilement abordable à un autre.

Nous tenterons tout de même d'isoler des améliorations graphiques qui, envisagées séparément, sont manifestement bénéfiques pour la lisibilité générale de graphes complexes. Nous verrons que ces améliorations sont parfois contradictoires, et que certaines ont plus d'effet positif lorsque l'on se trouve dans un paradigme d'affichage plutôt qu'un autre.

Enfin, et comme c'est le cas pour les techniques d'analyse statiques présentées dans le chapitre précédent, nous devrons nous limiter à celles d'entre ces techniques dont l'implémentation est envisageable dans les fait. En effet bon nombre de solutions sont subordonnées à la mise en œuvre d'algorithmes NP-complets (calcul d'une représentation planaire, optimisation de la longueur totale des arcs... ), ce qui nous amènera à mettre en œuvre des approximations plus ou moins efficaces de ces solutions.

### 3.2.2. Méthode en trois étapes : topologie – forme – métrique

Cette méthode a été mise au point par C. Batini et al. [BNT86] pour améliorer la lisibilité des graphes sous les conventions suivantes : arcs orthogonaux et sommets disposés sur une grille, avec la réduction des espaces vides comme facteur esthétique complémentaire. Ces conventions correspondent selon les auteurs à des besoins fréquents dans le domaine de la conception de systèmes informatiques. C'est par exemple le cas pour représenter les dépendance fonctionnelles entre les différents modules d'une architecture logicielle, ou lors de l'affichage du schéma d'une base de données.

Elle procède par raffinements successifs augmentant progressivement la lisibilité du graphe, tout en conservant certaines propriétés du graphe. Ainsi, les faces – qui, dans la représentation d'un graphe, sont les surfaces délimitées par des arcs

ou des segments d'arcs [1]– sont-elles contraintes à rester délimitées par les mêmes arcs. Pour ce faire, les auteurs définissent successivement trois classes d'équivalence de graphes aux contraintes de plus en plus strictes :

- Deux graphes ont la même topologie si l'on peut obtenir l'un en appliquant à l'autre une série de transformations conservant les faces du graphe et l'ordre des segments qui les délimitent.

- Deux graphes ont la même forme s'ils ont la même topologie et si seuls les longueurs de leurs arcs (ou segments d'arcs) varient.

- Deux graphes ont la même métrique s'ils ont la même forme et que l'on peut obtenir l'un en effectuant une composition de translations et de rotations de l'entièreté de l'autre.

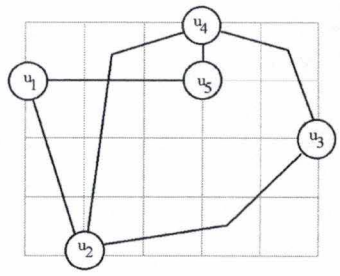

*Figure3.12*

La méthode consiste à créer des représentations successives du graphe tels qu'ils seront successivement équivalents au graphe optimal selon l'une des trois définitions ci-dessus.

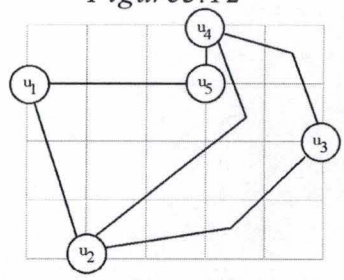

*Figure3.13*

La première étape consiste à assurer (ou si cela n'est pas possible à simuler) la planéité du graphe. (figure 3.13) On peut utiliser une méthode simplifiée consistant à établir un sous graphe maximal ou la planéité est garantie, puis en ajoutant les arcs supplémentaires un à un en prenant soin de minimiser les croisements avec les arcs existants, ces croisements étant quoi qu'il en soit remplacés par des sommets factices. Au terme de cette étape de « planarisation », le graphe aura la même topologie que le graphe terminal.

La seconde étape consiste à rendre les arcs orthogonaux, en s'assurant éventuellement que le nombre de segments reste raisonnable.

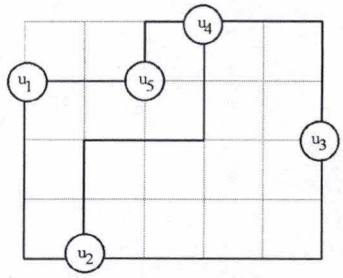

*Figure3.14*

L'ultime étape vise à compacter le graphe afin de réduire sa surface totale. Le graphe produit a la même forme que le précédent puisque ce compactage se contente de réduire, là ou cela est possible, la longueur des segments d'arcs sous la double contrainte de l'orthogonalité et du positionnement des nœuds sur une grille.

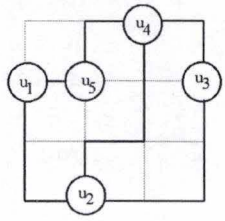

*Figure3.15*

La méthode est illustrée par l'exemple ci-dessus. Les quatre premiers dessins montrent les transformations du graphe générées par l'application de la méthode. On remarque les différentes équivalences. Le cinquième donne une solution

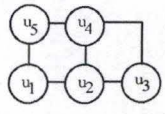

*Figure3.16*

[1] Si nous prenons l'exemple de la figure 3.14, trois faces sont définies : l'une délimitée par les arcs joignants les sommets $u_b$ $u_2$ $u_4$ $u_5$, l'autre délimitée par $u_2$ $u_3$ $u_4$ et la troisième face, appelée face externe, est la surface entourant le graphe.

plus optimale qui n'a pas été trouvée par la méthode. Cela soulève l'importance du placement initial qui n'est pas pris en compte par la méthode. Nous verrons que cette omission a un certain nombre d'avantages, en permettant notamment de faire intervenir des contraintes sur la position de certains sommets. En outre, cette méthode peut-être modifiée en y ajoutant des facteurs esthétiques comme la volonté de voir les arcs orientés aller de haut en bas, ou en réduisant le compactage à une seule dimension pour conserver une représentation reflétant une hiérarchie de rang.

### 3.2.3. Méthode hiérarchique sur les graphes acycliques

Présentée dans [DIBATTISTA99], cette méthode est adaptée à un paradigme (composition de conventions, de facteurs esthétiques et de contraintes) différent de la méthode vue au point précédent : le positionnement des nœuds d'un graphe acyclique sur une grille, la représentation de haut en bas des arcs orientés par des lignes droites et l'optimisation des croisements entre arcs. La méthode agit également par étapes successives :

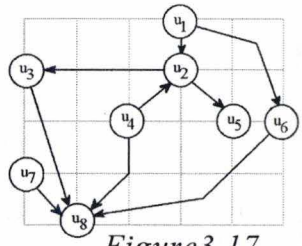

*Figure3.17*

- La première étape vise à établir la hiérarchie. (figure 3.18)

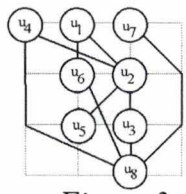

*Figure3.18*

- La seconde étape vise à réduire la longueur des arcs à l'unité. Cela est possible en insérant autant de sommets factices qu'il y a de niveaux traversés par chaque arc de longueur supérieure à l'unité. (figure 3.19)

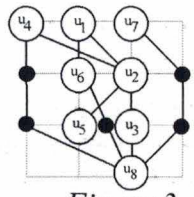

*Figure3.19*

- L'étape suivante consiste à réduire les croisements entre arcs. Sachant que les arcs ont tous une longueur de un, cette tâche peut être effectuée couche par couche. (figure 3.20)

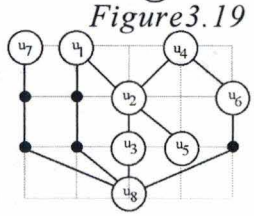

*Figure3.20*

- La dernière étape consiste en un compactage horizontal des sommets. Il faut noter que la contrainte des arcs en ligne droite n'est que partiellement respectée, car les sommets factices peuvent être interprétés comme des brisures dans les arcs de départ. En outre, on recourra a des contraintes supplémentaires pour déterminer le positionnement des nœuds dans les couches 'creuses', à savoir celles présentant moins de sommets que les couches contiguës. On songera par exemple à centrer un prédécesseur par rapport à ses successeurs, ou a le placer à la verticale de l'un de ses prédécesseurs ou successeurs. (figure 3.21)

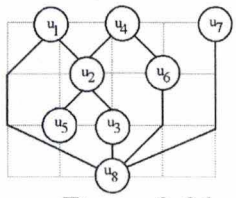

*Figure3.21*

On notera que comme pour la méthode précédente, l'ordre dans lequel les facteurs esthétiques sont abordés dans les étapes successives établit implicitement un classement d'importance de ces facteurs.

Cette méthode peut être généralisée aux graphes quelconques en envisageant une étape préliminaire pour extraire un sous graphe acyclique maximal du graphe de départ. Deux méthodes sont envisageables : la suppression d'un arc par cycle détecté, ou l'inversion du sens de l'un de ces arcs. Il est tout de même souhaitable que ce sous graphe soit très proche du graphe de départ afin de ne pas détruire le fruit de l'optimisation lors de la réinsertion des arcs qui n'appartiennent pas au sous graphe.

### 3.2.4. Généralisation

Des exemples qui précèdent, on peut déduire que les méthodes d'optimisation de l'affichage d'un graphe procèdent par transformations successives préservant les acquis des étapes précédentes. Ces étapes successives produisent des graphes vérifiant un nombre croissant de contraintes esthétiques. Dans ce contexte, plus les contraintes sont introduites tôt dans la séquence, moins elles sont limitées par des étapes précédentes, et donc plus leur influence sur le graphe est importante. C'est ainsi que la planéité ou la hiérarchisation apparaissent fréquemment comme premières étapes de l'optimisation.

Dans la suite de ce chapitre, nous verrons quels sont les choix judicieux de transformations dans notre situation particulière.

### 3.2.5. Une méthode « sur mesure » ...

Afin de définir précisément la méthode que nous allons appliquer, il convient dans un premier temps de répertorier les conventions, facteurs esthétiques et contraintes qu'il serait souhaitable de mettre en œuvre. Certaines découlent des caractéristiques des graphes à représenter alors que d'autres s'imposent comme pré requis aux étapes fondamentales.

#### 3.2.5.1. Représentation existante

Pour bien définir l'objectif à atteindre par la méthode, nous illustrons ci-dessous un exemple de représentations de l'environnement et du store tel qu'ils sont habituellement utilisés dans le projet JavAbInt.

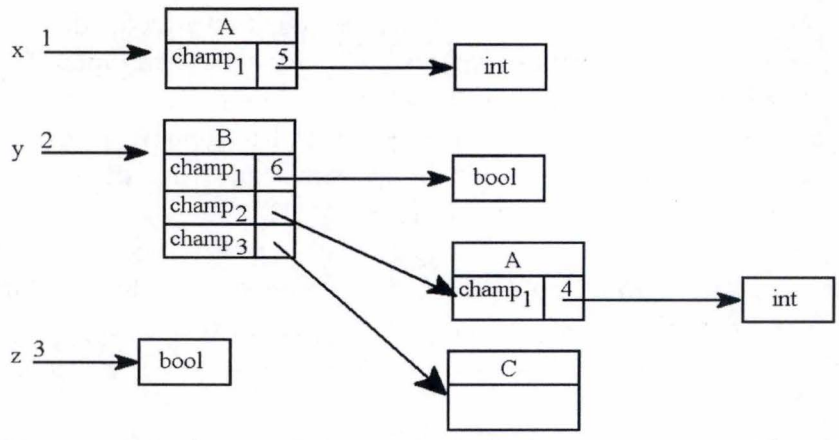


*Figure 3.22 : exemple de représentation de l'environnement et du store*

### 3.2.5.2. choix des conventions, facteurs esthétiques et contraintes

*conventions*

Au premier abord, nous constatons que les nœuds sont alignés sur une grille. De plus, l'orientation des arcs suit une dominante allant de la gauche vers la droite.

*Facteurs esthétiques*

Même si notre exemple est relativement simple, l'absence de croisements en facilite bien évidemment la lisibilité. C'est également le cas de l'alignement des nœuds.

*Contraintes*

La contrainte principale porte sur la présence des noms des variables à l'extrême gauche de la représentation.

*Aspects spécifiques*

Un des aspect primordiaux du graphe représenté ci dessus, et qui n'est pas directement pris en compte par les méthodes présentées précédemment est l'importante place occupée par les sommets. Nous avons considéré jusqu'ici que la représentation d'un sommet était en fait un point de taille plus ou moins négligeable. Cette approche n'est pas satisfaisante dans notre cas. L'espace important qu'occupe les sommets nous oblige à tenir compte d'éventuels chevauchements de ceux-ci ou de manière plus prévisible, l'existence d'arcs traversant ces sommets.

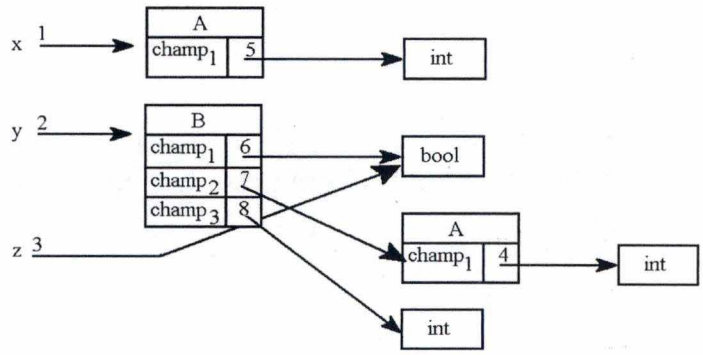


*Figure3.23*

Une autre problématique qu'il nous faut résoudre est liée au faut que les arcs ne sont pas connectés aux sommets aléatoirement. D'une part, les arcs entrant et les arcs sortants doivent se trouver sur des côtés opposés du sommet, d'autre part l'ordre des arcs sortants est intimement lié aux nœuds qu'ils atteignent mais aussi à l'ordre des champs au sein du nœud (tout au moins si l'on souhaite conserver l'ordre des champs dans les différents nœuds de même type) .

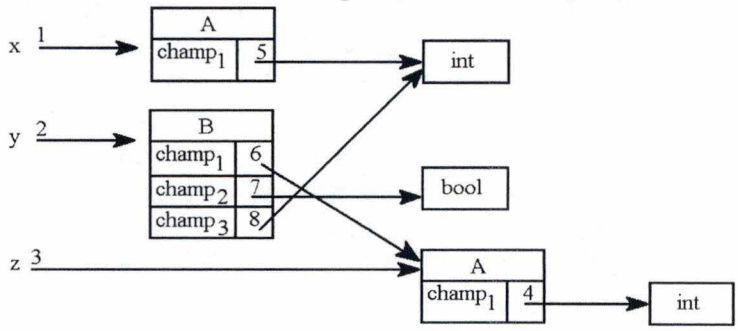


*Figure3.24*

Les conventions, facteurs esthétiques et contraintes à prendre en compte sont donc, par ordre d'importance, le positionnement des nœuds sur une grille, l'établissement d'une hiérarchie de rang (qui suppose la réduction à un sous graphe acyclique maximal) allant de la gauche vers la droite et plaçant les nœuds correspondants aux éléments du domaine de l'environnement à gauche, et la réduction des croisements entre arcs.

*Solutions spécifiques*

Une solution aisée à la première problématique serait de considérer que les arcs qui ont une portée de plus d'un niveau (les autres ne rencontrant pas de nœuds) suivent obligatoirement l'horizontale lorsqu'ils traversent une couche de nœuds.

On peut également utiliser la convention imposant aux arcs d'être composés de segments orthogonaux parallèles aux bords des nœuds. Cette dernière solution sera approfondie dans la suite de l'exposé.

### 3.2.5.3. Transformations du graphe

Nous donnons ici une succession de transformations d'une représentation graphique d'un couple <environnement, store> mettant en œuvre les solutions vues ci-dessus.

*Construction de la hiérarchie*

La première étape vise à mettre en place une hiérarchie sur les sommets. Les nœuds sont alors représentés par des points. Elle implique un phase préliminaire de suppression des cycles et est complétée par la génération de nœuds factices afin de décomposer les arcs en chemins d'arcs simples. (figure 3.25)

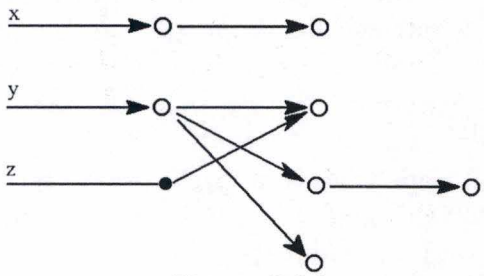

*Figure3.25*

*Réduction des croisements entre arcs*

Il ne s'agit pas ici de trouver une représentation planaire du graphe, mais plutôt de réduire autant que faire se peut le nombre de croisements entre arcs. Nous utilisons pour cela un algorithme relativement simple qui examine chaque niveau **n** de la hiérarchie (de gauche à droite sur la figure 3.25) et qui dégage une permutation des nœuds du niveau **n+1** minimisant le nombre de croisements d'arcs entre ces deux niveaux. (figure 3.26)

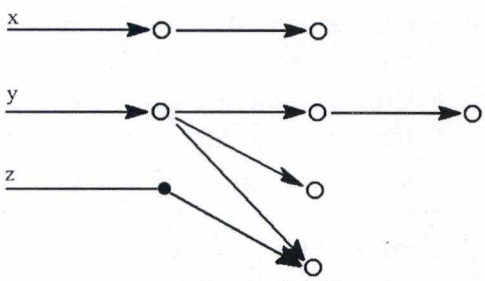

*Figure3.26*

*Orthogonalisation des arcs*

Les arcs sont orthogonalisés parallèlement aux bords de l'espace à deux dimensions. Deux cas de figure se présentent : soit l'arc compte un seul segment, soit il en compte trois. Dans ce cas, le premier et le troisième sont horizontaux (et interceptent respectivement les nœuds d'origine et d'extrémité de l'arc) alors que le second segment est vertical.

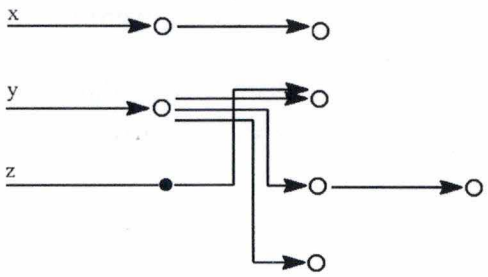


*Figure3.27*

*« Dilatation » des nœuds*

Les points qui symbolisent les nœuds sont remplacés par une représentation complète du nœud pour en faire les sommets que nous avons illustré plus haut (figure 3.22). Cette dilatation conserve la forme du graphe (cfr. Point 3.2.2) en ce sens qu'elle se contente d'allonger certains segments (figure 3.28).

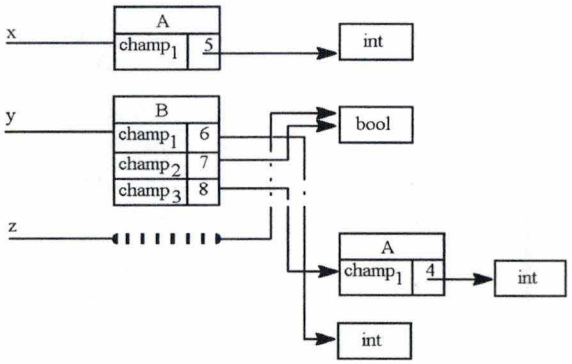


*Figure3.28*

Afin de maintenir le positionnement sur une grille, la largeur des nœuds est constante. L'écartement entre les couches horizontales est calculé pour chaque couche sur base du maximum des hauteurs des nœuds de la couche. Ainsi, dans la figure 3.28, la seconde couche à l'épaisseur du nœud de classe B qui est le plus grand de la couche.

Les nœuds factices sont dilatés en largeur. En fait, ils disparaissent totalement (voir figure 3.29).

Leur positionnement est légèrement différent des nœuds « réels ». En effet, il semble intéressant des les positionner entre deux couches horizontales puisqu'ils disparaissent dans la représentation finale en se confondant aux segments d'arcs qu'ils connectent. Or les arcs qui traversent plusieurs niveaux de hiérarchie ont avantage à passer entre les couches horizontales afin d'éviter les nœuds. Nous nous proposons donc de doubler le nombre de couches horizontales. Les couches impaires seraient réservées aux nœuds « réels » alors que les couches paires (grisées sur la figure 3.29) seraient occupées par les nœuds factice et donc par les arcs ayant une portée supérieure à l'unité.

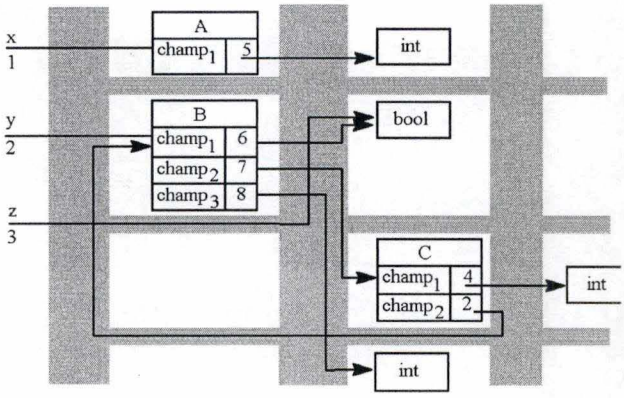


*Figure3.29*

Notons que si l'on extrait les couches paires et les interstices verticaux entre les nœuds (espaces grisés), on retrouve une représentation fort proche de celle de la figure 3.27. Ces transformations ne seraient pas totalement lisibles si l'on n'y ajoutait que les « couloirs » verticaux et horizontaux entre les nœuds doivent faire chacun l'objet d'un positionnement des segments qui les traversent afin d'éviter que plusieurs d'entre eux ne se superposent, ce qui rendrait le graphe difficile à lire.

Voilà qui résous le problème des arcs traversant inopinément la représentation des nœuds. Pour ce qui est des éventuelles intersections entre arcs dues à des points de connexions fixés (voir figure 3.24), nous préconisons la permutation des champs afin d'éviter ces croisements supplémentaires. Cela ne pose pas de problème particulier puisque les nœuds sont clairement caractérisés par le type de l'instance qu'ils représentent.

Le fait de devoir opposer d'un côté les arcs entrants et de l'autre les arcs sortant n'est pas pénalisant pour les arcs qui font partie du sous graphe acyclique qui sera établi dans la phase de hiérarchisation. Dans le cas des autres arcs, soit on les ajoute en recherchant le chemin le plus court, soit on peut envisager la recherche d'un chemin induisant un minimum de croisements. (cfr. L'arc allant de l'instance C à l'instance B dans la figure 3.29).

## 3.3 Description du problème

L'affichage d'un graphe tel que celui modélisant l'environnement et le *store* est, par la complexité de la structure de données à représenter, une tâche ardue dès lors que l'on s'intéresse un tant soit peu à la lisibilité du résultat. La démarche adoptée pour la mise en œuvre du module d'affichage dont il est question dans ce chapitre prends donc largement cet aspect en compte, notamment en implémentant la méthode décrite au point précédent.

Malgré la subjectivité liée à l'appréciation de la lisibilité d'un graphe, les paramètres (conventions, facteurs esthétiques et contraintes) choisis pour cette méthode aiderons à cette lisibilité.

Le graphe constitué par l'environnement et le *store* est à première vue un graphe orienté quelconque. Il est toutefois intéressant de s'assurer que certains sous graphes puissent apparaître comme des arbres, ou tout au moins voir la représentation graphique de leurs arcs orientés dans une même direction. C'est par exemple le cas pour les relations de composition entre une instance et ses champs (une instance est constituée de références à d'autres locations qui la composent). En outre, la réduction des croisements entre arcs, voir la planéité, sont des mesures qui facilitent la lecture du graphe. Ces contraintes seront définies en détail dans les points qui suivent.

Nous avons donc opté pour les conventions graphiques suivante :

- Les nœuds donneront une information sur le type de l'instance et sur les différents champs qui la composent. Ils seront placés sur une grille hétérogène (au niveau de l'espacement des horizontales).

- Les arcs seront de type ligne brisée avec des segments orthogonaux

Les facteurs esthétiques principaux sont :

- La minimisation du nombre de croisements entre arcs.

- L'utilisation d'une hiérarchie sur un graphe acyclique maximal extrait du graphe

En outre, une contrainte intéressante sur la solution graphique est d'imposer que les nœuds directement accessibles depuis l'environnement, et qui constituent les racines du graphe acyclique dont il est question ci-dessus, se trouvent rassemblés en bordure de la solution graphique, sur un même côté[2].Spécification macroscopiques

[2] Voir à ce sujet la représentation graphique de l'exemple introductif

Le module d'affichage prend en argument les objets modélisant un couple <environnement, store> et produit un composant graphique affichant cette structure de données.

Le composant graphique prendra la forme d'un panel Java. Outre l'affichage du graphe de manière « lisible », il est souhaitable d'offrir la possibilité à l'utilisateur de mettre en œuvre un certain nombre de transformations qui amélioreront également la lecture du graphe :

- Plusieurs niveaux de zoom peuvent permettre tantôt de visualiser l'entièreté du graphe, tantôt d'en détailler certaines parties.

- Il peut être intéressant de n'afficher que les instances référencées par une variable particulière de l'environnement. On peut également souhaiter réduire l'affichage selon d'autres critères logiques (hiérarchie basée sur la relation d'extension, sur l'origine des références liées à une location...)

- Un affichage incrémental peut être envisagé, permettant à l'utilisateur de développer l'affichage du graphe progressivement, par exemple en limitant la profondeur à un certain nombre de niveaux.

- Certains arcs remarquables méritent d'être mis en évidence. Il en va ainsi de l'arc reliant une entrée de l'environnement à sa location.

Enfin, on permettra à l'utilisateur d'obtenir des informations complémentaires concernant un nœud particulier, par exemple en utilisant une fenêtre « *popup* » affichées lorsque l'utilisateur clique dessus.

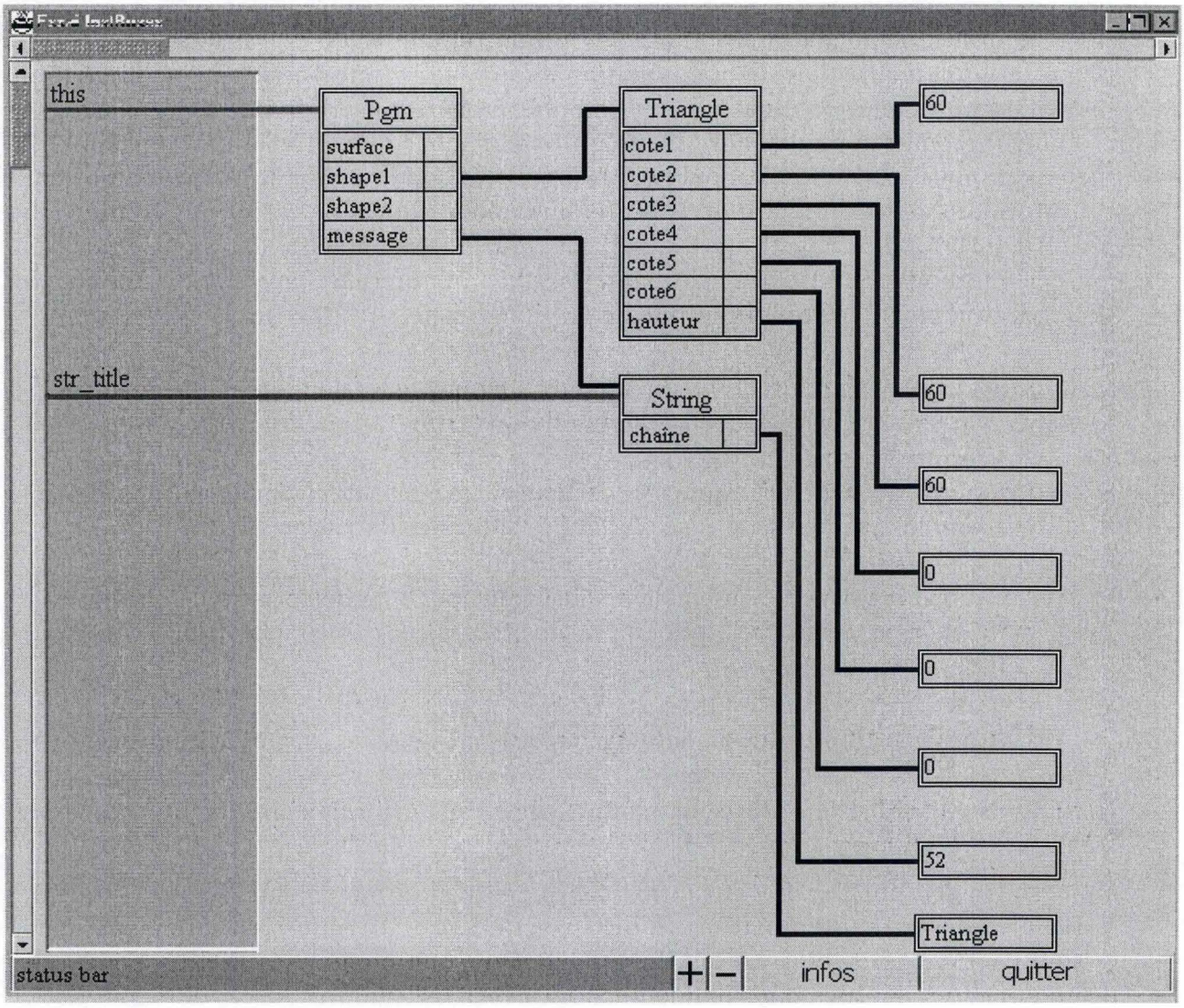


*Figure3.30 : maquette de représentation du couple <environnement, store>*

On notera que la maquette ci dessus présente un cas fort réduit. Ceci étant, un « simple » programme Java peut rapidement voir son *store* composé de plusieurs dizaines de locations. Il s'agit en fait de la transposition de la figure 3.2.

A la lumière de cet exemple, on comprends aisément la difficulté que peut constituer la lecture de pareil schéma pour peu qu'il ait quelque ampleur.

### 3.3.1. Architecture

Le module d'affichage que nous nous proposons de réaliser va mettre en œuvre une série de techniques de manipulation de graphes d'une part, et de manipulation d'interface graphique d'autre part. En outre, nous avons choisi de « préparer » les données en entrée pour rendre la structure à afficher plus proche des concepts liés aux graphes (arcs et sommets). Dans cette optique, plutôt que d'enrichir l'information existante (en dérivant de nouvelles classes de celles qui contiennent déjà l'information constitutive des nœuds du graphe) nous avons pris le parti de créer une nouvelle structure de données, qui fait référence aux entités du couple <environnement, store> plutôt que de les étendre.

Nous détaillons l'architecture du module selon une démarche « *top-down* ». La tâche y est décomposée en trois sous-modules distincts :

- Le premier créera une structure de donnée facilement manipulable, encapsulant l'ensemble des informations nécessaires à l'affichage du graphe.

- Le second regroupera l'ensemble des manipulations et traitements sur le graphe, avec entre autre le calcul de la position relative des nœuds du graphe selon la méthode définie au point 3.2.5.

- Le troisième affichera ce graphe et gérera les paramétrages propres à l'utilisateur.

#### 3.3.1.1. Spécification du sous-module de transformation de la structure de données

Ce sous module prendra en entrée les mêmes environnements et stores que le module dont il fait partie. Cette structure est définie formellement dans [POLLET99]. Les classes qui la modélisent sont communes à l'ensemble du projet et sont décrites dans le document [JAVABINT2000].

En sortie, le sous module retournera un « graphe logique » qui est un objet modélisant l'ensemble des arcs et nœuds. Il contient les information suivantes :

- Un ensemble de e-nœuds, encapsulant les éléments du domaine de l'environnement, c'est à dire les noms de variables.

- Un ensemble de i-nœuds représentant les instances.

- Un ensemble d'arcs représentant les références entre les nœuds.

Nous choisissons de représenter les nœuds par des objets encapsulant les informations suivantes :

- Une référence à l'objet sémantique représenté,
- L'objet graphique représentant l'entité décrite ci dessus,
- Un ensemble d'informations utiles pour les manipulations dont le graphe fera l'objet dans les sous modules suivants : rang au sein de la hiérarchie, identifiant de composante fortement connexe, paramètres de positionnement divers...
- Un ensemble de références modélisant les arcs : références aux champs de l'instance, le cas échéant, référence au vers l'instance mère,
- Un ensemble de références permettant le chaînage inverse des mécanismes précédents.

Nous donnons ci dessous une spécification plus formelle du sous module. Dans cette dernière, un i-nœud du graphe est un objet de classe EnvGraphNode, alors que le graphe lui même est de classe EnvGraph.

Objectif du sous module: sur base d'un environnement et d'un store, créer un graphe logique.

Paramètre d'entrée : un environnement et un store.

Conditions sur les paramètres d'entrées : l'environnement et le store sont des objets corrects par rapport à la sémantique du langage analysé.

Paramètre de sortie : un graphe, sous la forme d'un objet de type EnvGraph

Conditions sur le paramètre de sortie :

- Le graphe modélise l'environnement donné en entrée de manière complète et cohérente.
- si le paramètre d'entrée est omis, il ne faut pas retourner de résultat.
- un code spécial est prévu pour indiquer la survenance de problèmes inopinés (défaut de mémoire, erreur système... ).

Algorithme :

```
[Nom : Construction du graphe - Paramètres: (env , store)]
    graphe = { }
    graphe = Créer un nœud du domaine de l'environnement (inst. courante(env),graphe )
    graphe = Développer le graphe (inst. courante(env), graphe )
    Pour chaque variable v_i ∈ env
        graphe = Créer un nœud du domaine de l'environnement (v_i, graphe)
        Si ∃ noeud ∈ graphe : location(nœud) = location(v_i)
            Alors
                Créer un arc (e-noeud (v_i) , noeud )
            Sinon
                graphe = Développer le graphe (v_i, graphe)
```

Structure du sous module :

Nous savons que le graphe à créer est composé de nœuds de 2 types : d'une part les e-nœuds, représentant des éléments du domaine de l'environnement, d'autre part les i-nœuds représentants des instances.

De plus, nous savons que chaque nœud de la première catégorie est directement associé à un et un seul nœud de la seconde, et que chaque nœud de la seconde catégorie peut référencer un ou plusieurs autres nœuds. En outre, pour chaque nœud de la seconde catégorie, il existe au moins un chemin entre un nœud de la premier catégorie et ce nœud, et que toutes les racines du graphe sont de la première catégorie.

Nous choisissons de réduire le store affiché aux locations qui peuvent être atteintes depuis l'environnement courant. Les autres éléments du store sont moins utiles à priori, et alourdiraient fatalement la représentation graphique.

Il est donc intéressant d'une part de générer – via un processus itératif sur les éléments du domaine de l'environnement – la partie de graphe comportant les nœuds de première catégorie, et d'utiliser d'autre part un processus récursif pour créer les nœuds de la seconde catégorie. Ces deux parties sont détaillés dans les paragraphes qui suivent.

*Sous module « Eléments du domaine de l'environnement »*

Le processus réalisé par ce sous module est la création des nœuds associés aux variables appartenant au domaine de l'environnement. Le traitement de chacune de ces variable va impliquer ...

- La création d'un élément graphique associé à ce nœud

- Si la location ne représente pas un type de base, la vérification d'un sharing éventuel sur une instance déjà traitée.

- L'établissement de l'arc entre le nœud modélisant l'élément du domaine de l'environnement, et l'instance associée, qu'elle soit préexistante ou, le cas échéant, qu'elle ait été crée via un appel au sous module décrit ci-dessous.

Objectif du sous module: sur base d'un élément de l'environnement et du graphe partiellement construit, créer un nœud correspondant à l'instance référencée par l'éléments de l'environnement passé en paramètre. En outre, la création de ce nœud déclenche de manière récursive la création des nœuds associés à ses champs.

Paramètre d'entrée : un élément du domaine de l'environnement et un graphe de type EnvGraph.

Conditions sur les paramètres d'entrées : l'environnement et le store sont des objets corrects par rapport à la sémantique du langage analysé. Le graphe contient des e-nœuds correspondant aux éléments du domaine de l'environnement qui ont déjà été incorporés dans le graphe.

Paramètre de sortie : le graphe mis à jour.

Conditions sur le paramètre de sortie : Le graphe en sortie est le graphe en entrée, incrémenté d'une modélisation de l'élément du domaine de l'environnement donné en entrée.

*Sous module « Elément du domaine du store »*

La tâche réalisée par ce sous module est la création d'un nœud associé à un élément du domaine du store. En outre, cette fonctionnalité prévoit un appel récursif sur les nœuds qui sont eux-mêmes référencés par le nœud traité (i.e. les champs de l'instance courante qui n'ont pas encore fait l'objet d'une « traduction » en un nœud du graphe).

Plusieurs étapes sont nécessaires à la réalisation de cet objectif : En premier lieu, bien sur, créer et initialiser l'instance de l'objet décrivant le nœud, et référencer les autres objets avec lequel le nœud interagit. On en profite également pour créer l'objet graphique associé au nœud, car l'ensemble de l'information utile à cette tâche est disponible.

Il faut noter que l'instance représentée peut également ne pas comporter de champs, ou être un élément d'un 'type de base' du langage, auquel cas le nœud sera une feuille du graphe. Les deux situations – feuille ou nœud « intermédiaire » sont décrites ci dessous...

*nœud représentant une instance*

La subtilité de ce sous module est de s'assurer de la bonne transposition du *sharing* dans le graphe. Il faut dans ce cas gérer le processus récursif de manière sélective afin d'éviter la création de doublons.

*nœud représentant une location d'un type de base*

Il n'y a pas la même subtilité ici. Il faut cependant noter que nous avons pris la liberté, pour clarifier le graphe, de bannir tout sharing sur les valeurs de base. Ce faisant, nous évitons l'affichage d'un certain nombre d'arcs convergents vers les feuilles, ce qui rend le graphe plus lisible sans représenter une entorse trop grave à la sémantique du langage.

Objectif du sous module: sur base d'un élément du store, créer le nœud correspondant à l'instance de ce dernier et déclencher de manière récursive la création des nœuds associés à ses champs, ou faire le lien si l'instance existe déjà.

Paramètre d'entrée : un élément du store et un graphe de type EnvGraph.

Conditions sur les paramètres d'entrées : -

Paramètre de sortie : le graphe mis à jour.

Conditions sur le paramètre de sortie : Le graphe en sortie est le graphe en entrée, incrémenté d'une modélisation de l'instance associée à élément du store donné en entrée.

#### 3.3.1.2. Spécification du sous-module d'optimisation de l'affichage du graphe

Ce sous module travaille sur le graphe tel qu'il est produit par l'étape précédente. Le résultat est un graphe dont les nœuds ont été enrichis d'informations relatives au positionnement des nœuds dans l'espace à deux dimensions.

Nous présentons ici une implémentation par étapes successives, comme le suggère la méthode présentée au point 2.3.5.

- La première va détecter et écarter les cycles du graphe. Au terme de cette phase, nous serons en présence d'un graphe acyclique plus facile à manipuler. Les arcs écartés ne sont pas supprimés pour autant, ils sont justes identifiés par une propriété spécifique.

- La seconde, relativement triviale, consiste à calculer le rang de chaque nœud afin de constituer une hiérarchie. Cela signifie que, dans la restriction du graphe générée par a phase précédente, un nœud sera positionné à un niveau supérieur à tous les nœuds du sous graphe acyclique qui le référencent.

- La troisième, visera à minimiser le nombre d'intersections entre arcs en permutant les nœuds de même rangs. La méthode prévoit également une phase

d'orthogonalisation et de dilatation des nœuds qui sont implicites dans l'implémentation.

*Sous module de détection et suppression des cycles du graphe*

Cette première phase est basée sur un algorithme simple de détection des cycles. Pour chaque nœud, on marque l'ensemble de ses successeurs. Pour chaque élément de cet ensemble, on tente de trouver un chemin qui mène à nouveau au sommet de départ. Si tel est le cas, un cycle a été mis en évidence. On brise les cycles éventuels en écartant l'un des arcs. Les arcs écartés seront réintroduits dans le graphe au terme de l'optimisation, au moment de l'affichage.

*Sous module de découpe en niveau - tri topologique*

Le principe du tri topologique est le suivant : les nœuds sont initialisés avec un indice nul. Les racines du graphe forment initialement l'ensemble des nœuds actifs. L'algorithme consiste à examiner chacun des nœuds actifs, et à gratifier l'ensemble des successeurs de ce dernier d'un poids strictement supérieur à celui de leur nœud père. Si ces successeurs ont déjà un poids supérieur à la valeur du père augmentée de un, ils la conservent.

L'opération est répétée jusqu'à ce que l'ensemble des nœuds à traiter soit épuisé, sachant qu'un nœud de cet ensemble le quitte lorsque il a été traité, c'est à dire lorsque tous ses successeurs ont vu leur indice mis à jour, et que l'ensemble est étendu progressivement à ceux d'entre les nœuds successeurs qui ont vu leur indice modifié (y compris des nœuds qui ont déjà été traités auparavant, mais qui sont garni avec un rang plus élevé).

La terminaison en un temps fini est garantie par l'absence de cycles, suite à l'étape précédente, et au nombre fini de nœuds. Chaque nœud pouvant prendre une valeur maximale plus élevée que le nombre de nœuds, et donc être réactivé un nombre fini de fois.

*Sous module de positionnement dans l'espace à deux dimensions*

Dans un premier temps, il est souhaitable de positionner les racines de ce graphe acyclique, à savoir les nœuds représentant des éléments du domaine de l'environnement. Cela est réalisé en calculant un « facteur de proximité » entre toute paire d'éléments du domaine de l'environnement, et en tentant de rapprocher les élément ayant la meilleure « proximité ».

Intuitivement, deux racines seront d'autant plus proches qu'elles partageront un plus grand nombre de successeurs. On adoucira toutefois cette intuition en considérant que plus un nœud est éloigné des racines (i.e. plus il a un rang élevé dans la hiérarchie), moins il aura d'influence sur la proximité des racines dont il est successeur.

Dans un second temps, on optimisera la position des nœuds à l'intérieur de chaque niveau. L'idée est que l'on peut trouver une permutation des nœuds d'un niveau **i** dans la hiérarchie qui minimisera les croisements des arcs entre le niveau **i** et le niveau **i – 1**. Pour ce faire nous allons transformer le graphe de façon à ce que les arcs aient une portée maximale d'un niveau. De cette façon, tous les arcs entre deux niveaux successifs entreront en ligne de compte pour la minimisation des croisements. Pour restreindre la portée des arcs, il suffit de créer des sommets intermédiaires fictifs à chaque niveau.

#### 3.3.1.3. Spécification du sous module d'affichage des composants graphiques

Ce sous module est d'autant plus simple que l'essentiel du travail est réalisé par les étapes précédentes.

D'une part, la représentation graphique des nœuds est fixée par le premier sous module. D'autre part, le positionnement des nœuds est défini par le second.

L'enjeu du sous module se trouve essentiellement dans l'affichage optimal des arcs entre les nœuds. Il faut en effet éviter une série de travers qui nuiraient à la lisibilité du résultat :

- Les arcs trouvent leur origine à droite du nœud origine et leur extrémité à gauche du nœud terminal
- Il faut veiller à ce que deux ou plusieurs segments d'arcs ne se superposent pas
- Au delà du processus de réduction des croisements entre arcs, et dans le cas de niveaux relativement « creux » (i.e. présentant un plus faible nombre de nœuds que les autres), on tentera de centrer un nœud par rapport à ses nœuds adjacents du niveaux suivant)

L'optimisation des « couloirs » entre les nœuds, permettant d'éviter la superposition des segments d'arcs se fait de la façon suivante :

- Chaque couloir possède initialement une seule « voie », à savoir qu'un seul arc peut être représenté.
- Les arcs sont ajoutés un à un. Lorsque un arc ne peut être ajouté sans induire une superposition, on ajoute une voie au couloir.

En outre, c'est au niveau de ce sous module que les possibilités d'interactions purement graphiques seront envisagées : zoom, affichage partiel, détails supplémentaires à propos des nœuds...

# Conclusion

Dans ce document, nous avons abordé deux aspects particuliers d'un projet de plus grande envergure : le projet JavAbInt. Ce dernier a pour but l'application des techniques de l'interprétation abstraite au langage Java afin de tenter de réduire l'inefficacité induite entre autre par le mécanisme de *dynamic binding* propre aux langages orientés objets. Concrètement, le projet vise à la mise en œuvre d'un analyseur statique générique de Java.

Les aspects abordés ont été d'une part la mise en œuvre d'un analyseur travaillant sur un sous langage restreint de Java, d'autre part le développement d'un outil de visualisation des couples <environnement, store> que ce soit au niveau de la sémantique concrète ou à celui de la sémantique abstraite.

L'étude de l'analyseur nous a permis de préciser les étapes d'une analyse par interprétation abstraite : définition du langage et de sa sémantique, définition d'une sémantique abstraite et d'un domaine abstrait, preuve de la correction de l'abstraction. Ces étapes ont ensuite fait l'objet d'une implémentation en Java mettant en œuvre l'algorithme multivariant.

La mise en œuvre de l'outil de présentation de l'environnement et du store nous a amené à mettre en œuvre des techniques de présentation relevant de la théorie des graphes. Nous avons ensuite établi un ensemble de propriétés auxquelles la représentation du graphe devait répondre. Nous avons également présenté quelques méthodes d'optimisation correspondant à des représentations courantes de graphes dans un espace à deux dimensions. A l'aide de ces méthodes, nous avons défini une méthode adaptée à la catégorie de graphes que nous étudions. Cette méthode a ensuite été appliquée à des exemples simples.

Cette seconde partie a principalement été réalisée lors d'un stage réalisé au sein de l'équipe du projet JavAbInt. Elle a été enrichie par la suite par de nouvelles sources d'informations. En conséquence, l'implémentation réalisée durant le stage ne prends pas en compte l'ensemble des observations présentées ici. Ceci étant, l'outil de visualisation implémente globalement la méthode définie dans ce mémoire, sans en présenter tous les raffinements. Il faut également signaler que comme tout processus de modélisation graphique, cet outil a fait intervenir une bonne part de subjectivité dans les choix esthétiques qui ont été posés.

En ce qui concerne l'analyseur par interprétation abstraite, il a constitué pour nous une première mise en pratique de ces techniques, et est donc probablement entaché d'erreurs de jeunesses. Il nous semble toutefois intéressant d'avoir pu mener à bien une analyse de bout en bout, fut-elle basée sur des simplifications drastiques. Même si l'intérêt d'un tel analyseur pour le projet global est relativement limité, son utilité pédagogique ne doit être négligée. En outre, les difficultés rencontrée lors de sa mise en œuvre sont autant de signaux dont il faudra tenir compte pour les développements futurs de l'analyseur générique qui constitue l'objectif central du projet.

L'outil de visualisation par contre est un premier pas dans la mise en œuvre d'un environnement graphique aisément abordable pour les utilisateurs du futur analyseur. A ce titre, il ouvre bien des possibilités de développements ultérieurs : que ce soit pour la création d'un émulateur 'pas à pas' permettant de tracer l'état du système au fil des instructions et de leur pendants abstraites, ou dans un systèmes de détection d'erreurs 'à la volée'.

L'avenir du projet est assuré car dans un futur proche, deux étudiants vont venir renforcer le staff 'permanent'.

# Bibliographie


[POLLET99] POLLET Isabelle. « Sémantiques operationnelles et domaines abstraits pour l'analyse statique de Java », *Facultés Universitaires Notre-Dame de la Paix, Namur – Belgium, Institut d'informatique, Memoire de DEA (Diplôme d'Etudes Approfondies)*, Septembre 1999

[DIBATTISTA99] DI BATTISTA G., EADES P., TAMASSIA R., TOLLIS I. *Graph Drawing, algorithms for visualization of graphs,* Prentice Hall, 1999

[INFO3105] LE CHARLIER Baudouin. Notes du cours « Interprétation Abstraite »,*Facultés Universitaires Notre-Dame de la Paix, Namur – Belgium, Institut d'informatique*, janvier 2000

[LECHARLIER99] LE CHARLIER Baudouin. « Définition du langage Vas-T'y-Frotte », *Facultés Universitaires Notre-Dame de la Paix, Namur – Belgium, Institut d'informatique, Notes de cours,* Mars 1999

[PLCC2000] POLLET I., LE CHARLIER B., CORTESI A. « Abstract Domains for Type Analysis of Java Programs», *Facultés Universitaires Notre-Dame de la Paix, Namur – Belgium, Institut d'informatique,* mai 2000

[FICHEFET99] FICHEFET Jean. « théorie des graphes», *Facultés Universitaires Notre-Dame de la Paix, Namur – Belgium, Institut d'informatique, Notes de cours,* janvier 1999

[COUSOT77] COUSOT P., COUSOT R., « Abstract Interpreattion : A unified lattice Model for static analysis of programs by construction or approximation of fixpoints ». *Conference record of fourth symposium on programming language, 238-252, Los Angeles,* Janvier 1977

[BNT86] BATTINI C., NARDELLI E., TAMASSIA R., *A layout algorithm for data flow diagrams*, IEEE trans. software enginrering SE-10 N°4, 1986

[JAVABINT2000] LE CHARLIER Baudoin et al. « définition des classes pour l'implémentation de la sémantique opérationnelle » *document de travail,* Octobre 1999

[HHN2000] HAYEZ C., HENDRICKX P., NOBEN K. « Interprétation abstraite d'un sous langage de Java », *travail réalisé dans le cadre du cours INFO3105 "Interprétation Abstraite" du professeur B. Le Charlier*, Juin 2000

# Annexe

*Définition d'un pseudo langage pour la génération d'un arbre syntaxique*

Nous donnons ici une description de la grammaire de ce pseudo langage dont le seul but est un gain de temps pour l'encodage de programmes de tests. Bien que très concise, celle-ci est fort proche de la syntaxe abstraite et peut d'ailleurs être traduite en arbre syntaxique en une passe et sans mémoriser d'états.

Le langage est structuré en tokens. Chacun d'entre eux représente un nœud de l'arbre syntaxique et occupe une ligne distincte.

Chaque token débute par un numéro identifiant le concept syntaxique qu'il décrit. L'utilité de ce numéro est de déterminer à coup sur la nature de l'objet syntaxique à créer. Nous en donnons la table ci dessous.

| Id. | *Descriptif* | Id. | *descriptif* |
|---|---|---|---|
| 01 | *Déclaration de champ* | 08 | *Instruction de fin de méthode (return)* |
| 02 | *Déclaration de variable* | 09 | *Déclaration de constructeur* |
| 03 | *Affectation* | 10 | *Déclaration de méthode* |
| 04 | *Appel de constructeur* | 11 | *Déclaration de la méthode Main* |
| 05 | *Appel de méthodes* | 12 | *Déclaration de classe* |
| 06 | *Alternative (if then else)* | 13 | *Déclaration de la classe main* |
| 07 | *Appel au superconstructeur* | 14 | *Déclaration de programme* |

Les tokens sont lus un à un. Vu l'absence de toute vérification, ils est important que l'ordre de leur succession soit rigoureusement celui prévu. Ainsi, un bloc d'instruction doit toujours être représenté dans l'ordre, et le token associé à la méthode dont ce bloc constitue le corps doit se trouver immédiatement après la dernière instruction du bloc. Nous donnons également, à titre d'exemple, la structure de quelques tokens :

01 String champ1

Ce token définit un champ (code 01) de type String et identifié par champ1.

05 4 v variable$_1$ v variable$_2$ method$_1$ v variable$_3$ v variable$_4$ END 5

Ce token définit un appel de méthode (code 05).

v est un flag signifiant que la chaine qui suit correspond à une expression du type variable. Lorsque le flag prends la valeur c, il s'agit d'un champ. En outre, lorsque le flag est une majuscule, cela signifie que l'expression a déjà été déclarée et qu'il faut donc utiliser la zone mémoire définie précédemment et non en recréer une autre.

variable$_1$ est l'expression qui sera affectée du résultat de la méthode.

variable$_2$ est l'expression sur laquelle se fera la méthode.

methode$_1$ est le nom de la méthode.

Variable$_3$, etc. Sont les paramètres effectifs. Leur nombre n'est pas vérifié.

5 est le point de programme terminal de l'instruction, tout comme en était le point initial.

13

Le code 13 est celui de la définition de la classe main. Comme cette définition n'est présente qu'une et une seule fois dans un programme, et que les champs et méthodes qui y sont liées doivent être définis auparavant, aucun argument n'est nécessaire.